UNIVERSITY OF OSLO
Department of Informatics

Dynamic Selection of Message Carriers in a Sparse and Disruptive MANET

Jan Erik Haavet

November 7, 2011

Acknowledgement

I would really like to thank my supervisors: Morten Lindeberg for his insight, support, guidance, discussions and excellent feedback. Vera Goebel for her support and guidance, and for always presenting me with clear and insightful feedback. You have both inspired me and helped me perform my best in this work. Thank you.

I would also like to thank my family and friends for their moral support, especially in these last few months.

Jan Erik Haavet
University of Oslo
Nov, 2011

Abstract

During an emergency and rescue (ER) operation, like an earthquake, we often have no or only partially available communication infrastructure. One viable solution to enable data communication is to use a mobile ad hoc network (MANET). However, MANETs in ER scenarios are often disruptive in nature due to physical obstacles, distance and sparse node density. Therefore, we need delay tolerant networking (DTN) with a store-carry-forward mechanism.It is realistic to assume that some nodes travel between the network partitions, hence act as carrier nodes. However, assuming *a priori* knowledge about which nodes can act as carriers is unrealistic.

In this master thesis, we analyze the application scenario and related work, and argue that a prediction-based approach for detecting carriers fulfills our requirements. We design a mechanism that provides dynamic selection of message carriers (DSMC) for DTN solutions. This is accomplished by that nodes in the network calculate, maintain and exchange delivery probabilities with all other nodes. Through this exchange, nodes learn which node has the highest probability of delivering the packets to the destination, hence act as carrier nodes. The design is implemented in NS3, together with the Dts-Overlay system, which is an ongoing development in the DT-Stream project [24] to tackle network disruptions through an overlay. We evaluate DSMC and compare its performance against a carrier selection strategy, called Static-Dts, which relies on *a priori* knowledge of carrier nodes. Through extensive simulations, we show that DSMC detects and utilizes carrier nodes for packet delivery. The performance of DSMC is nearly as good as that of Static-Dts, in terms of packet delivery. DSMC induces only a limited increase in delay, due to the dynamic selection of carrier nodes. Since the *a priori* knowledge required by Static-Dts is unrealistic, we argue that this increased delay is an acceptable trade off for a realistic solution. Through an analytical model, we show that the overhead of DSMC is low and scales well when the amount of nodes in a sparse MANET increases. For dense MANETs, this overhead potentially represents a problem of scalability, however these are not the kind of networks we are addressing.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Background and Motivation

In emergency and rescue (ER) scenarios, such as an earthquake or a train accident, efficient and reliable communication between rescue personnel is crucial for the efficiency of the rescue effort. This includes coordination between different units, e.g., police, fire department, health personnel and overall organizers.

Portable, two-way radio transceivers are the traditional communication tool in ER scenarios and can certainly be used, but they support only communication in the form of audio via radio. They do not support services such as Audio Visual (AV) services, e.g., video streaming. They also suffer from a range limitation as the range, typically a few kilometers, is not long enough to cover large or wide spread scenarios like an earthquake.

There are other ways of maintaining communication between rescue personnel that support other forms of communication. This includes cell phones and existing Wi-Fi networks. These networks however, require a working infrastructure, which can not be guaranteed in an ER scenario. In urban environments, infrastructure might be damaged or overloaded by users. In more rural environments, the infrastructure might be non-existing or blocked by obstacles, e.g., a mountain. The ER scenario may also spread over several locations, or be in a location with physical obstacles, like tunnels, making these communication technologies insufficient by themselves.

The need to support other forms of communication in the ER scenario, and not just audio via radio, is useful for more complex applications. This

includes applications like multimedia streaming, file transfer and event notification. As an example of multimedia streaming, rescue personnel could wear head mounted cameras, or cameras could be set up to cover the disaster/accident area. This would allow the organizers of the rescue effort to get a better overview, so they can make more qualified decisions.

In order to stream multimedia data, we need devices capable of transferring data packets, either directly with each other or using other devices as relays. One plausible solution is to have devices, with computational capabilities, spread throughout the scenario area that can use routing protocols to route data packets between devices.

The use of mobile ad-hoc networks (MANET), is a promising solution to providing data communication in such a scenario. A MANET does not rely on any existing infrastructure and can be created ad-hoc by mobile devices carried by rescue personnel. However, MANETs have many challenges to deal with in ER scenarios. The layout of the disaster area, including distances, obstacles like buildings etc., and movement of mobile nodes can lead to network partitions within the MANET. Such partitions are not supported by any of the standard MANET routing protocols, e.g., OLSR, AODV or DSR, but much research has gone into developing new protocols and adapting old protocols to the domain.

In the research community, MANETs that are spread out and suffer disconnections are called sparse or disruptive MANETs. To solve the disconnection problem that such disruptive MANETs suffer, many solutions have been suggested. These solutions fall under the larger research area of delay tolerant networks (DTN). DTN are networks in which disconnections are expected and cover more than just disruptive MANETs, like deep space communication, certain sensor networks and mobile communication under extreme conditions, e.g., under water. Most DTN protocols implement a store-carry-forward (SCF) [33] paradigm, in which designated carrier nodes can store and carry data between partitions to allow communication despite network disconnections. The SCF mechanism is often implemented using an overlay network on top of already existing routing protocols.

Utilizing message carriers for the carry operation requires knowledge about which nodes are suited for carrying messages between partitions. Much of the research on DTN assumes that these carriers are predefined. This assumption is not always realistic, as for an ER scenario where any member of the rescue effort could be moving between network partitions. Even if some nodes were known to be carrier nodes in advance, any solution relying on

this would probably not utilize other nodes that also move regularly between partitions.

1.2 Problem Statement

In an ER scenario, using a MANET to maintain data communication is a viable solution. However, because of the disruptive nature of a MANET in such a scenario, we need a store-carry-forward mechanism in order to support delay tolerant routing. Using an overlay network is one way to implement this mechanism. Assuming *a priori* knowledge about which nodes can act as carriers between network partitions is unrealistic. A better solution would be to instead discover these carriers dynamically. In this master thesis, we aim to develop a mechanism that provides this discovery of carrier nodes to an overlay routing protocol. The plan is to let nodes gather and utilize each others delivery probabilities, in order to discover carrier nodes.

1.3 Outline

In this master thesis, we start by describing the application scenario and domain in Chapter 2. We identify the requirements of an AV streaming service in a disruptive MANET and ER scenario. We present an ongoing development for such a service, the Dts-Overlay, and identify the requirements for a dynamic selection of message carriers (DSMC) mechanism. In Chapter 3, we introduce the required technologies for streaming in a MANET. We analyze related work in Chapter 4 in search of ideas that can help us design the DSMC mechanism. In Chapter 5, we describe the design of DSMC and discuss issues relating to that design. We discuss the implementation of DSMC in Chapter 6 and show how it is implemented together with the Dts-Overlay. We evaluate the performance and overhead of DSMC in Chapter 7. Our conclusions from this master thesis are given in Chapter 8 along with a discussion of future work.

Chapter 2

Application Domain

In this chapter we introduce our application scenario (see Section 2.1) from DT-Stream [24], a project aiming to enable audio visual (AV) streaming over heterogeneous, mobile and disruptive MANETs. Section 2.2 lists the requirements from DT-Stream and related requirements of an AV streaming service in an ER scenario We describe Dts-Overlay system [16] in Section 2.3, which is an ongoing development in the DT-Stream project to tackle network disruptions through an overlay. Based on the application scenario, and what has already been solved in the Dts-Overlay, we analyze and find key requirements in Section 2.4 that are targeted in this master thesis, i.e., a mechanism for dynamic selection of message carriers (DSMC).

2.1 Application Scenario

For this work we consider an emergency and rescue (ER) scenario like the one found in [26], where there has been a train accident in a rural environment. The accident may be caused by different events, such as: (1) A collision between two trains, (2) sabotage of the train, (3) technical difficulties and (4) natural disasters like a landslide or an earthquake. Trains can carry a large amount of people and the potential for death and injuries in a train accident is high. The accident area might be hazardous for anyone involved due to weather, fire or the accident itself. This will cause an acute need to transport passengers, especially those injured, away from the accident area and to a safe location. In this scenario, the safe location is a cabin, about a kilometer away from the accident. This is also the location of the command

and control center (CCC) where resources, rescue personnel and supplies are gathered. The CCCs main purpose is to serve as a central place for those governmental departments, e.g., firemen, police or doctors, to organize their resources from.

ER operations require a lot of coordination and communication among rescue personnel. AV services, like a video stream from rescue workers wearing head mounted cameras could be useful in order to provide better information to those coordinating the effort at the CCC. Engineers, doctors and organizers could watch these video streams in order to make faster and more qualified decisions, e.g., an engineer could watch a video stream of the train wreckage and quickly alert rescue personnel about possible impending hazards. Most of the devices needed to support AV services exist today. This includes head mounted cameras, lightweight hands-free microphones and mobile devices with wireless and computational capabilities. However, connecting these devices through a network is still a challenge, because we cannot assume that there is any infrastructure available. The infrastructure near the rescue area might be destroyed by the accident, a natural disaster or it might be non-existing due to the location of the accident. E.g., in this scenario, there is no infrastructure, because the train accident is in a rural location.

One possible solution to the lack of infrastructure is to use a mobile ad-hoc network (MANET). We assume, for this rescue scenario, that all rescue personnel wear a communication device, such as a mobile phone, with wireless communication capabilities and that it is possible to set up these devices so that they comprise a MANET. In addition, we assume that the rescue personnel is spread out to attend to different tasks and that the MANET therefore is sparse in nature. An example of such a MANET in this scenario can be seen in Figure 2.1.

Time is a valuable resource in an ER scenario and so each device should require minimal configuration, e.g., not require rescue personnel to configure the devices to make it work optimally. We assume that each device communicates with any other device that the rescuer is wearing, such as a head mounted camera.

MANETs, especially in challenging environments such as ER scenarios, are unreliable and topologies might change quickly. There are several issues: (1) Packet loss due to collisions and interference in shared media transmissions, (2) transient connections between devices due to the mobility of the personnel and lowered transmission range due to physical obstacles like ter-

Figure 2.1: Example of Nodes Forming a MANET in a Possible ER Scenario

rain. As a result, any solution for networking, i.e., protocols etc., should adapt to changes quickly and should not assume full network connectivity.

In fact, in this type of ER scenario, a MANET is likely to have network partitions. We can expect network partitions to be formed around the actual accident, composed of working rescue personnel. And we can expect another partition to be formed around the CCC, composed of organizers, preparing rescue workers and health personnel. It is likely that these two partitions will not be able to communicate directly, because the CCC is set up at a safe distance away from the accident location. We therefore assume that the MANET, in addition to being sparse, is disruptive in nature. In order to support data communication, like AV streaming, between partitions, e.g., nodes in the rescue area partition and nodes in the CCC partition, we need a solution for delay tolerant networking (DTN).

Some rescue personnel move between partitions with regularity. Their job is to transport injured people, carry supplies and equipment back and forth between the accident area and the CCC. This movement is not random and can be utilized to transport messages between network partitions.

The use of mobile devices in such a scenario implies some resource limitations. This includes battery power, storage space and processing power. Particularly battery usage should be minimized. It is unrealistic and unpractical for rescue personnel to switch batteries during the operation. If one device is essential for maintaining communication between network partitions and that device runs out of energy, then the ability of the network to provide delay tolerance could be severely limited or in the worst case lost.

2.2 DT-Stream Requirements

DT-Stream is an ongoing research project [24] aimed to develop new solutions that enable AV streaming services over heterogeneous, mobile and unstable networks in ER operations. As we describe in the application scenario, there are many issues to consider when designing a solution. We now outline the three main goals of the DT-Stream project and list related requirements of an AV streaming service in an ER scenario.

1. A delay tolerant streaming application that does not break when network partitions occur, but instead adapts its functionality, and seamlessly proceeds, when connectivity is back.

    - **Delay tolerant:** It is probable that a MANET in an ER scenario has network partitions. Transferring messages between these is a challenge and is not supported by the standard MANET routing protocols. The solution should therefore be delay tolerant in the way that it should support delay tolerant routing of packets.
    - **Utilize non-random movement:** Rescue personnel in an ER scenario do not move randomly and it is likely that some will move between network partitions with regularity. This movement can be used to deliver packets between otherwise disconnected network partitions, like message ferries.
    - **Best effort video streaming:** To make use of AV streams, packet delivery should be maximized to avoid a "choppy" video experience and the delay of the video stream should be minimized to give the user the most up-to-date information.

2. A self-adaptive overlay that caches AV data at selected nodes to increase the resilience and performance of the AV service.

    - **Assume no infrastructure:** An ER scenario might take place in a location with little or no existing communication infrastructure. The infrastructure might also be overloaded, damaged or destroyed. Any communication solution should therefore not assume that such infrastructure exists.
    - **Resilient and fault tolerant:** MANETs, particularly those that are disruptive in nature, suffer from packet loss, transient connec-

tions, lowered bandwidth and range. The solution should therefore be resilient and tolerate these issues.

- **Reactive to changes:** As the MANET changes, connections and routes status change. The solution should react to these changes and try to use the most up to date routing information available.
- **No configuration:** A rescue workers priority should be to focus on his assigned task in the rescue operation, not to configure a device for optimal settings. Time is essential in an ER scenario and devices should therefore require no configuration other than turning the application on.

3. Autonomic resource management to discover, monitor and manage resources through distributed admission control and multi-path routing protocols.

    - **Optimize resource usage:** Mobile hand-held devices rely on limited resources such as battery, storage and processing power. For instance, should the battery run out on a potentially critical device in the communication network, the networks ability to maintain communication between nodes could be severely limited. It is therefore important to optimize the resource usage of each node.

2.3 Dts-Overlay

One ongoing development in DT-Stream for communication in an ER scenario, as described in the previous section, is the delay tolerant stream (Dts)-Overlay system [16]. It is inspired by MOMENTUM [5], an overlay solution for transferring multimedia over sparse MANETs. Overlay networks are logical networks built on top of another network. In this case, the overlay network is built on top of the network layer, to be more specific the OLSR routing protocol. In MOMENTUM, certain nodes called session nodes, implement the overlay network and uses it to support delay tolerance. The overlay at these nodes implements a store-carry-forward (SCF) mechanism where nodes can store packets, which have no route to the destination, carry them and eventually forward them to the destination node. In MOMENTUM, if a node wants to send a packet to a node that is outside its network

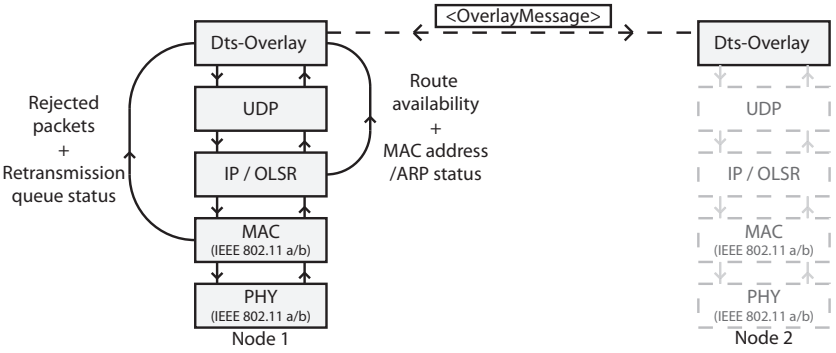

Figure 2.2: Cross-layer Information Flow in Dts-Overlay (Lindeberg et al. 2011).

partition, it replicates the packet to all session nodes in its partition. This way, it is more likely that one of these session nodes will meet the destination nodes network partition and forward the packet. In Dts-Overlay, all nodes are session nodes, as all nodes implement the overlay network. As opposed to MOMENTUM, Dts-Overlay does not use replication to forward packets. Instead, packets are forwarded to predefined carrier nodes, which move between specific network partitions. Each carrier only moves between two network partitions, in terms of the ER scenario, this is between the CCC and the area of the accident.

2.3.1 Stack Architecture

In Dts-Overlay, the authors build the overlay network on top of UDP, the OLSR MANET protocol and IEEE 802.11. As seen in Figure 2.2, the overlay uses the OLSR routing table, via a cross-layer mechanism, to find a route to a destination and the address of the next hop in that route. The destination address is added to an overlay message header and added to the packet. Then the packet is sent to the next hop in the route using the UDP transport protocol. This forces each node in the routing path to handle each packet and allows them to alter the routing path with potentially more up-to-date information (hop-by-hop routing).

2.3.2 Route Awareness

If a node does not find the destination node in its OLSR routing table, it looks up a local cache of recent routes. If a recent route suggests a next hop that is within range, the packet is forwarded to this node. The idea here is that the next hop node on the recent route to the destination is probably closer to the destination, and this in turn will lower the delay of packets. If no recent route can be found, the packet is stored in a local buffer. It is stored in the buffer until a new path emerges, or until a carrier node is within range. In Dts-Overlay, designated carrier nodes move between the network partitions. These carrier nodes can be identified by their unique IP address range. When a node meets a carrier node, it forwards its packets, which the carrier node stores and carries until it has a routing path to the destination node.

2.3.3 MAC Support

The MAC layer in IEEE 802.11 mainly handles packet loss as random drops due to transmission collision, i.e., it is handled by up to seven retransmissions. In disruptive MANETs, this behavior is often counter productive as loss is often due to nodes no longer being within range. In Dts-Overlay [16], authors remedy this by introducing MAC support, which aims to reduce the amount of retransmissions and avoid dropping packets. The Dts-Overlay uses a cross-layer mechanism to interact with the MAC layer (see Figure 2.2).

In Dts-Overlay, the MAC layer does not drop the packet after it reaches the retransmission limit. Instead, the packet is transferred up to the overlay and buffered there. When the overlay is sending a packet, it first finds a next hop address as described, but before the packet is sent to this next hop address, the overlay checks if the retransmission queue to that address is filling up. If it is, this indicates that there is something wrong and the packet should be buffered instead. This limits the amount of unnecessary retransmissions. Some packet loss also occurs due to the use of the address resolution protocol (ARP), when its reply is lost. To remedy this, Dts-Overlay assures that the address resolution process is successfully finished before sending packets.

2.4 Requirements Analysis

Dts-Overlay fulfills many of the requirements listed from the application domain. It does not **assume any infrastructure** and is built using a MANET. It uses an overlay network, which implements SCF, to support **delay tolerance**. By using the MAC support functionality, the authors makes the overlay more **resilient to faults** such as inaccurate routing table data, link quality and transmission collisions. The overlay **utilizes the non-random movement** of predefined carrier nodes to deliver messages between network partitions. Each packet is forced up to the overlay at each hop in the path, and the route is updated based on the information at this node. The idea is that this allows the routing to **react faster to changes** and use more up to date information. In Dts-Overlay, using replication is avoided so there is little overhead from this protocol, which supports **minimized resource usage**. Using the MAC support functionality, fewer packets are dropped and the SCF mechanism combined with the forwarding strategy attempts to minimize the delay. Both respond well to the requirement of **best effort video streaming**. Although the overlay fulfills many of the requirements, there are some open issues in Dts-Overlay.

One issue is the assumption of *a priori* knowledge about which nodes act as carrier nodes, which violates our requirement of requiring **no configuration** to work optimally. This assumption is not practical, because it requires the personnel wearing the devices to reconfigure it every time he or she changes their role in the rescue effort, e.g., a worker transporting injured to the CCC, stops at the CCC to do other work. It is unlikely that some rescue workers only move between the two partitions. Reconfiguring a mobile device can not be a prioritized task for rescuers in such a scenario. Another issue is that these predefined carriers are limited to carrying packets to one destination, in this case one node in the CCC partition. The authors do not consider how Dts-Overlay would handle more partitions with different receivers. Due to these issues, there is a need for Dts-overlay to automatically discover and select carriers, to different destinations, and adapt to appearing/disappearing carriers without human intervention.

Additional Requirement

Based on Dts-Overlay and the issues outlined, there is one obvious requirement that Dts-Overlay is not addressing. This is; **Dynamic selection of**

message carriers:. Assuming *a priori* knowledge about carrier nodes is unrealistic and unpractical. Nodes in the network should therefore try to discover and select the best message carriers for the destination of its messages.

In the following chapter, we study related work to discover ways in which a dynamic selection of message carriers can be performed. We address this requirement again in Chapter 5, where we look at the design of DSMC, which is our solution to support this requirement.

Chapter 3

Background

This chapter introduces background and key terms and technologies referred to throughout this master thesis. We start by describing wireless communication in Section 3.1. MANETs and routing protocols are described in Section 3.2. Section 3.3 describes challenges faced for MANETs when there are network partitions and Section 3.4 describes how such challenges are normally addressed. Finally, in Section 3.5, we describe multimedia streaming over MANETs.

3.1 Wireless Communication

The IEEE 802.11 is a set of standards developed by the Institute of Electrical and Electronics Engineering (IEEE) for implementing wireless local area networks (WLAN). It is the basis for the implementation of Wi-Fi wireless networks. Wi-Fi networks operate in the 2.4GHz (802.11.b, 802.11.g) and 5GHz (802.11.a) unlicensed radio bands and can theoretically provide up to 11 Mbps (b), 54 Mbps (a,g). The range of Wi-Fi devices depend on their antenna, radio and the environment in which it is used, e.g., in open landscape the range can be up to 300 m in the right conditions, while inside a building it could be limited to 30-40 m.

Communication is established between nodes with Wi-Fi capabilities. Nodes communicate with each other and with access points (AP) using radio waves on specific frequency bands. There are two main modes of operation. The first, infrastructure mode, is often set up for homes, offices and businesses that have an AP. The AP controls communication between nodes and route

messages to other networks, like the Internet. The other mode of operation is ad-hoc mode, which does not require an AP but instead requires nodes to communicate directly with each other. This is often used for more transient and disruptive networks, such as a MANET. The main difference between these two modes is the AP, which routes all messages. In ad-hoc mode, all nodes share the responsibility of routing messages in stead of relying on the AP.

Wi-Fi is widely implemented in all sorts of devices. This includes sensors, cell phones, personal digital assistants (PDA), laptops, personal computers and many more.

3.2 Mobile Ad-Hoc Networks

A Mobile ad-hoc network (MANET) is a self-configuring network consisting of heterogeneous mobile nodes that communicate using Wi-Fi in ad-hoc mode. More specifically, MANETs are used when there is no infrastructure, i.e., no access point (AP) available, and when the nodes are mobile in nature. In a regular Wi-Fi network, APs perform routing between nodes in the network and possibly other networks connected to the AP. To support wireless communication without APs, MANETs use the ad-hoc mode which enables each node to act as both client and router, i.e., in addition to sending and receiving their own packets, nodes are also responsible for routing packets to other nodes. Each node keeps links to other nodes within that nodes radio range. These links are transient because of the mobility of the nodes. Therefore routing protocols from wired networks, which assumes long lasting links, are not applicable to MANETs. The main challenge of MANET routing protocols is to maintain accurate routing information with minimal overhead.

MANET routing protocols

Nodes in a MANET communicate either directly with each other or using other nodes as routers. Finding routing paths between nodes is the job of routing protocols. MANET routing protocols perform routing in two main ways, proactive and reactive. In a proactive protocol, the goal is to have a route ready when needed. Which means that the routing protocol needs to regularly exchange information with other nodes to keep changing routes up

to date. In a reactive protocol, routes are calculated on-demand, adding a delay to get a connection started but significantly lowering the overhead of the protocol.

DSR

Dynamic Source Routing (DSR) [12] is a reactive routing protocol. A route discovery packet is broadcasted from the source. Each node receiving such a packet adds its address to the packet and rebroadcast the message, but only if it has not already done so for the same source/destination/sequence number combination. When the route discovery packet reaches the destination, it is returned to the source using the accumulated addresses and the route is formed.

AODV

Ad hoc On-Demand Distance Vector Routing (AODV) [23] is another reactive protocol. When a route is needed, it broadcasts a route request packet. Each node rebroadcasts the packet if its sequence number is larger than the last sequence number received for that source/destination combination. It then updates its local routing table so that this node has a pointer to where the packet came from. When the request packet reaches the destination, the destination node sends back a route reply packet on the same path. Each node on the path registers where the packet came from in their local routing table, and forwards the packet back towards the source. When it reaches the source, each node on the path knows which node to forward it to, if a packet is headed either to the source or the destination.

OLSR

The Optimized Link State Routing (OLSR) [9] is a proactive protocol. Each node uses HELLO messages to discover its 1-hop, 2-hop neighbors and to perform a distributed election of a set of multipoint relays (MPR). MPRs are elected so that every 2-hop path between two nodes go through a MPR. These MPR nodes then exchange Topology Control (TC) messages, sharing information about which nodes each MPR covers. Topology information is flooded often enough that routes are not wrong for long periods of time. When a node needs a route, is simply looks in its routing table, which stores the next hop for each destination.

3.3 Delay Tolerant Networks

The discussed MANET routing protocols assume a certain amount of connectivity between nodes. For scenarios where the node density is low, there are no guarantees that all nodes are connected. In fact, for some MANET scenarios, disconnection and network partitions are expected. For these scenarios, the mentioned MANET routing protocols are not working. The MANET routing protocols do not consider any possibility of delay tolerant routing, which would be needed in the presence of network partitions.

Thus the need for delay tolerant networking (DTN) is established. DTN include networks in which there might be long periods of disconnection for some nodes and partitions which are never directly connected. Examples of some environments where DTN might be needed include military, disaster response, sensor networks, inter-planetary communication and disruptive MANETs. Supporting delay tolerance requires adaption of existing MANET routing protocols or new routing protocols. We investigate some of these types of protocols more closely in Chapter 4. There are two methods of supporting DTN, the first is to use an overlay network with a store carry forward (SCF) [33] mechanism where nodes are able to store messages and carry them until they can be forwarded to the destination. Another method is to alter the IP protocol and make it delay tolerant [21].

3.4 Overlay Networks

One common technique for supporting DTN is the use of overlay networks. An overlay network is a virtual network built on top of another network. Links between nodes may be virtual links, so instead of a physical link between two nodes, a link can represent multiple physical links, possible future links, remembered links etc. The same principle applies to the nodes in an overlay network. They can represent one or more physical devices or a complete network. As an example, peer-to-peer networks are overlay networks because the nodes are only logical nodes. One computer may be three such logical nodes in the peer-to-peer network. One link between two peers does not represent a physical connection, as there can be any number of physical connections and networks between the two peers.

Although overlay networks can be useful, they may also introduce overhead. This overhead could be one or more of several types of overhead.

- **Computation:** Cost of computing routes, probabilities, etc. This overhead is low in its self, but the total overhead depends on the frequency in which it is run.
- **Bandwidth:** Cost of adding extra headers to packets or sending extra packets. The more bandwidth used for the overlay network, the less bandwidth is left for sending packets.

3.5 MANET Multimedia Streaming

Video streaming over the Internet is an important and well-established service. Examples of services using video streaming include video conference, news, entertainment and sports broadcasts. With the development of laptops, smart phones with Wi-Fi and computational capabilities, the possibility and need for multimedia streaming has also arisen for MANETs. Much research effort has gone into providing multimedia streaming over MANET. Some common techniques are listed below.

- Replication and caching to provide disruption and partition tolerance.
- Cross-layer optimization. Includes using multipath routing and multiple description video encoding to provide sufficient video quality.
- Packet prioritization at the MAC layer.

These techniques, and variations of them are discussed in more detail in [17].

Chapter 4

Related Work

Many research efforts focus on delay tolerant networking (DTN). They differ in scenario assumptions, required knowledge and techniques used for supporting DTN. In this chapter, we look at this related work to find techniques and ideas that could help us meet the requirements for a dynamic selection of message carriers. We start by presenting an overview of the related work in Section 4.1. We divide the many related works in to two main categories which we study in detail in Section 4.2 and 4.3. Some unsuitable related work is presented in Section 4.4. Finally, we summarize our findings from the related work in Section 4.5.

4.1 Overview

We have classified some of the categories in which the papers differ in order to get a better overview of the related work.

- Parameter from related works scenario:
  - **Mobility Model:** From assuming random waypoint mobility to some reappearing routes to static routes with the possibility of proactive movement.
- Techniques used to support DTN:
  - **Replication:** Ranging from one copy of each packet to one copy per node in the network.

- **Knowledge:** From assuming each node has no knowledge of the network, to assuming partial or full knowledge of some sort. This could include topology from a routing protocol, node contact probabilities, which nodes move and act as carrier nodes, etc.
- **Routing decisions:** Made at source (source routing), at key nodes or per hop (hop by hop routing).
- **Prediction technique:** How the protocol predicts future contact or a contacts probability of delivery to a certain destination. This is not relevant if the work does not rely on prediction.

In Table 4.1, we have listed some of the work we have studied related to routing in DTN. They are sorted by their relevance to the stated requirements from Chapter 2, with mobility model being the most important factor. Most of the work can be split into two main categories or a combination of the two. These are **replication-based protocols** and **prediction-based protocols**.

According to our requirements, we are looking for work which utilizes non-random movement, does not require unrealistic *a priori* knowledge and that makes its routing decisions per hop. We are also looking for ideas which can help detect carrier nodes dynamically.

Table 4.1: Related Work Categorized

Related work	**Mobility model**	**Replication**	**Knowledge**	**Routing decisions**	**Prediction**
CAR[20]	Reappearing routes	No	Topology	Per hop	Contact duration and node mobility
MEED[13]	Reappearing routes	No	Topology	Per hop	Expected delay
Cluster[10]	Nodes with similar mobility patterns	No	Inter-cluster topology	Source and gateway	Contact frequency per time unit
PRoPHET [18]	Reappearing routes	Yes	No	Per hop	Contact frequency, Transitive

MEDD[7]	Reappearing routes	No	Contact statistics gathering period	Source	Expected dependent delay
Dominating set[25]	Reappearing routes	No	Dominating set	Per hop	Contact duration
MaxProp[3]	Reappearing routes	Yes, limited by time	No	Per hop	Contact frequency
Leapfrog[32]	Reappearing routes	No	Predefined contact probabilities	Per hop	Predefined contact probabilities
SARP[11]	Reappearing routes	Up to L copies, spread by delivery probability	No	Per hop	Contact frequency per time unit. Transitivity and time decay
MV[4]	Reappearing routes and reactive mobility	1-2 copies	No	Source	Region contact frequency per time unit
OPF[19]	Long term regularity	2^H copies, with max H hops	Mean inter-meeting times for all nodes	Per hop	Mean inter-meeting time and number of remaining hops
MobySpace [15]	Reappearing and static routes	Up to N copies at source	Nodes mobility pattern	Per hop	Frequency of locations
Hybrid[8]	Static clusters and proactive message ferries	No	Cluster topology	Source and gateway	No

Message Ferry[33]	Proactive message-ferries/nodes	No	Message ferry routes	Per hop	No
TPDR[6]	Static routes with small deviations	No	Movement model for all nodes	Per hop	Calculated from Movement model
SEPR[28]	Random waypoint in grid	No	Stochastic network topology	Per hop	Contact duration and Shortest expected path
Spray and Wait[27]	Close to random	L copies	No	Per hop	No
Epidemic[29]	Random	Max 1 copy per node	No	Per hop	No

4.2 Replication-based Protocols

One method often used to support DTN, is message replication. Generally, the goal of these methods is to spread a message so that it is robust to node failures, has as low delay as possible and with as little overhead as possible. Most of these proposed protocols assume close to random waypoint mobility. Replication-based schemes generally create a lot of contention for both the limited buffer space at each node and network capacity. Most of the work using replication-based protocols focus on limiting the replication without lowering performance.

4.2.1 Epidemic Routing for Partially-Connected Ad Hoc Networks

Vahdat and Becker [29] had one of the early suggestions for supporting DTNs. They present a routing protocol for DTN called Epidemic Routing. The protocol relies on each node spreading each message to all new node contacts that do not already possess a copy. Upon node contact, two nodes exchange a summary vector of the messages it currently has. This way, a node discovers if the other node has some messages that it does not have, and if the remaining buffer space allows it or the messages are considered more important than those already stored, it requests these messages from the other node. The authors also introduce a hop count field in each message, allowing users of the protocol to limit the replication of each

message. A hop count of 1 will only allow for direct delivery, while 2 hops allows for 1 relay before direct delivery, etc.

This protocol leads to a large amount of overhead, but given large enough message buffers at each node, it also provides near optimal delivery rate and delay. This is especially true for random waypoint mobility scenarios, where mobility patterns can not be utilized.

4.2.2 Spray and Wait: An Efficient Routing Scheme for Intermittently Connected Mobile Networks

Spray and Wait [27] is a flooding based protocol, like the Epidemic routing protocol. The authors aim to limit message replication without lowering delivery rate and increasing delay. This is accomplished by bounding the number of copies and transmissions per message. It scales very well to larger networks, because of this bounding and requires less transmissions per node as the network size increases. Spray and Wait also includes analytical methods to compute the number of message copies that is needed to achieve an average message delivery delay.

The spray and wait protocol consists of two phases:

1. Spray: For each message from a source, L messages are copied, spread and forwarded to L distinct relays.

2. Wait: If the destination is not found in the spray phase, copies are stored until a direct contact to the destination can be established.

The authors suggest several ways of spraying. One simple way is for the source node to copy the message to the first L distinct nodes it meets. This is simple but not optimal and so they suggest another method, which they call binary spray and wait. It works as follows: Any node with $N > 1$ message copy that encounters another node, transfers $N/2$ copies to that node. When a node is left with only 1 copy, it must get a direct contact to finish its transfer.

4.3 Prediction-based Protocols

In most networking scenarios, nodes tend to have greater probabilities of meeting certain nodes, or visiting certain areas. In prediction-based DTN protocols, store-carry-forward(SCF) is optimized by utilizing non-random movement. Each work finds some way of expressing the contact probability between nodes, or a variation, e.g., the delivery probability when forwarding using a specific node. Most of this work uses a single message copy, but some additionally use replication.

4.3.1 Probabilistic Routing in Intermittently Connected Networks

PRoPHET [18] was one of the first works to use contact prediction in order to provide a better SCF service in a DTN. Earlier work in DTN focused on the Epidemic protocol and ways in which to effectively limit message replication. Replication is not excluded in PRoPHET, but it is not the focus of the protocol.

In PRoPHET, each node maintains a list of destinations and its delivery probability for each of them. When nodes meet, their respective values for each other is updated, as well as transitive contacts, e.g., delivery probability from node A to C through B. The probability value for direct contacts is estimated at each contact, where meeting nodes increase their probabilities for each other and all other contact probabilities are lowered. This makes this strategy rely on encounter frequency, which does not necessarily reveal the true contact probability. As discussed in several articles [25, 11, 19], nodes moving together will only increase their probabilities once, while in fact they are continuously connected. Another issue with the encounter frequency strategy is to consider two nodes A and B barely within communication reach of each other. Such a connection could result in an on/off fluctuating state, which would increase contact probability even though the connection is rather unstable.

When nodes have finished exchanging delivery probabilities, they check if the other node has a higher probability of delivering any of the messages found in its buffer. If so, these are forwarded but not deleted. The node keeps the message as long as there is room in its buffer. When it meets a new node, it will go through the same routine. This way, there can be some replication, limited by buffer size, at each node.

4.3.2 Practical Routing in Delay-Tolerant Networks

Jones et al.[13] aimed to implement a DTN routing protocol that is self-configuring, well performing in different connectivity patterns and does not waste buffer and network resources. They focus on minimizing end-to-end delay, which reduces the amount of time messages occupy buffer space, which in turn should minimize messages dropped from buffer overflow. Authors propose the metric; minimum estimated expected delay (MEED), which is based on computing expected waiting time using observed contact history. To compute MEED, a node records contact durations over a sliding history window. The size of the window is left as a configurable parameter. Choosing a low size will enable it to change quickly, which can be good to discover changes and bad for fluctuating nodes. A larger window size will avoid fluctuations but might miss short lasting connection opportunities.

Nodes spread MEED to other nodes it meets. This is done by implementing MEED into a link state protocol, which is distributed in an epidemic fashion. This allows a new node to start routing after only one contact with any given node.

Routing decisions are made at each hop based on the local link state table. Instead of routing a message as it arrives, the authors instead argue that deciding the next hop is best to do for every contact a node makes as it will take advantage of more up to date routing information. This will require more processing resources as routes will be recomputed often and may limit the scalability of this protocol.

4.3.3 Adaptive Routing for Intermittently Connected Mobile Ad Hoc Networks

Musolesi et al. [20] introduces Context-aware Adaptive Routing (CAR), which is a general framework for the evaluation and prediction of context information. Its aim is efficient and timely delivery of messages in a delay tolerant MANET. Instead of using only contact prediction, CAR aims to allow other factors to affect routing decisions.

CAR is built on top of a link state protocol, a simplified version of DSDV, which is used in inter-partition routing and to exchange delivery probabilities. Where there is no direct contact to the destination, CAR uses the carriers with the highest probability of delivery, which is calculated from the context.

They define the context as a set of attributes that describe the aspects of the system that can be used to optimize the process of message delivery. Since they assume a proactive routing protocol, every node periodically sends both the information related to the underlying synchronous routing (in DSDV this is the routing tables with distances, next hop host identifier, etc.), and a list containing its delivery probabilities for the other nodes.

There can be an arbitrary number of attributes in their context. These attributes can be combined and given different weights to adjust for their significance. Attributes can also be excluded/included during run time. This allows for a dynamic approach to predicting delivery probability. Instead of only using one attribute, e.g., contact frequency or mobility pattern, this approach allows for a good combination of different attributes.

In their simulations, they only use two attributes:

1. The rate of change in connectivity. Estimated by periodically calculating the percentage of a node's neighbors that have changed their connectivity status.

2. Probability of being in the same partition as the destination node. To calculate this, they periodically check the underlying routing table and run a

Kalman filtering process [14] which will yield a value [0,1] that represents this. This process has the advantage that it does not need to remember connectivity history. It only needs one value, the current probability, which is updated each period by the Kalman process.

4.3.4 Clustering and Cluster-based Routing Protocol for Delay Tolerant Mobile Networks

Dang el al. [10] proposes a clustering protocol for a delay tolerant MANET. It is well known in the research community that clustering improves network scalability [1] and the authors claim that this is the first work looking at clustering in a DTN. Clustering in a DTN is challenging because of the lack of continuous connections between nodes. Nodes have inconsistent information and therefore respond differently, which leads to less stable clusters. The authors use contact probabilities between nodes to form clusters and select cluster gateways, because of the lack of continuous connections. These contact probabilities are calculated using an exponentially weighted moving average (EWMA). The authors claim that the EWMA converges towards the true contact probability. The EWMA is calculated by updating each contact probability at each time slot, increasing it if a node has been met in that time slot or decreasing it if it has not. As opposed to PRoPHET, this method uses frequency per time unit and avoids the issues found in using just contact frequency.

Routing of messages is performed like regular cluster protocols. In the cluster, routing is done per hop using the cluster topology table at each node. Inter cluster messages use the designated gateways.

This protocol does not rely on any underlying MANET routing protocol, and instead aim to be a replacement for it.

4.4 Unsuitable Related Work

There are several other works listed in Table 4.1, however they are not considered in detail here because of different assumptions or techniques that are not appropriate for the stated requirements. In this section, we briefly explain why the related works are not considered in more detail:

- In MEDD [7], authors rely on a contact statistic gathering period in order to make their contact prediction reliable. In our scenario, we can not assume such a time period as routing could start at any time. In addition, their protocol makes routing decisions at the source node and as discussed this is not optimal.

- Samuel et al. [25] introduces a DTN MANET protocol that uses a virtual network topology where each link reflects the probability of future contact. In order to provide optimal routing, they rely on a stationary node that gathers statistics, calculates a network dominating set and distributes this set to all nodes using an epidemic protocol. Both the need for a special and stationary node, and the delay in calculating and distributing the dominating set are factors which do not fit the scenario requirements.

- In MV [4], authors assume some nodes move proactively to support routing where it is needed in the network. Although robots, vehicles or people moving proactively to support the routing effort, is not unrealistic in our ER scenario, it is not included in our scenario specification.

- In MaxProp [3], the authors use contact frequency to predict future contacts between nodes. However, as stated earlier when discussing PRoPHET [18], this has several drawbacks as it can not accurately reflect actual contact probabilities.

- Xiao et al. introduces Leapfrog[32], which focus on optimal opportunistic routing in probabilistically contacted DTN. They rely on predefined unchanging contact probabilities between nodes, which is unrealistic and limited in its use, especially for an ER scenario.

- Elwhishi et al. [11] introduces a self-adaptive routing protocol (SARP) for delay tolerant MANET. Although it is similar to CAR in using contact duration to predict future contact between nodes, it does not provide the opportunity to add more attributes to this prediction value. SARP does not utilize any topology, but instead relies on replication. (As a node meets another node, it gives it $J < L$ of its L copies according to that nodes delivery probability to the destination.)

- In [28], authors introduce shortest expected path routing (SEPR). They use contact duration and shortest expected path to calculate delivery probabilities, which are exchanged upon contact. Although interesting, this work assumes nodes follow grid lines in a random fashion, which reduces the effect of predicting future contacts.

- Leguay et al. [15] relies on nodes knowing where they are. Then they calculate node mobility pattern and route messages after the probability of nodes going to certain locations. This has some issues, such as needing location information and that it does not consider nodes that visit the same location may in fact never visit them at the same time.

- The optimal probabilistic forwarding (OPF) [19] protocol relies on the *a priori* knowledge of mean inter-meeting times between all nodes. This assumption does not work for MANET scenarios, where such knowledge is impossible to know in advance.

- In [8], authors propose a hybrid routing protocol for DTN. They assume stationary clusters with message ferries providing inter-cluster routing. They also assume message ferries work in static routes with the possibility to proactively move between some nodes according to traffic conditions. Another issue with this work is that it requires node set up in advance, which is not ideal for a MANET in an unpredictable ER scenario. Similarly, in [33, 6] they assume message ferries move in well known nearly static routes.

4.5 Summary

Vahdat and Becker [29] introduced epidemic routing which has served as the basic idea for using replication to support DTN. In [3, 11, 15, 27, 19] work is focused on reducing the overhead introduced when using replication. In general, replication seems to be one solid approach to supporting a DTN. However in choosing replication, users of it accept high overhead in order to achieve low delay, which does not fit our resource optimization requirement. Most of the work relying on replication does not utilize the predictability of non-random movement, which we also listed as a requirement.

Utilizing predictability and non-random node movement is the focus of many papers [18, 13, 20, 10, 7, 25, 4, 28] and it has been shown that prediction/probability of packet delivery can indeed be used to improve routing in DTN. The most important requirement in Chapter 2, which is not supported by the Dts-Overlay is to find a dynamic method of utilizing the non-random movement of nodes between partitions to deliver messages. Most of the papers that utilize predictability fulfill this requirement. However, not all of them fit as well with the Dts-Overlay system. Both MEED and CAR fit well with Dts-Overlay. However, CAR also contains ideas for how a more dynamic delivery probability value can be estimated. In MEED, the delivery probability is based only on contact duration between nodes.

Chapter 5

DSMC Design

In this chapter, we present the design of our mechanism called dynamic selection of message carriers (DSMC). The purpose of DSMC is to improve one aspect of the store-carry-forward (STC) mechanism in a delay tolerant network (DTN), i.e., the selection of carrier nodes. It is motivated by the requirements of the application domain. We start by summarizing the assumptions from Chapter 2 and design considerations in Section 5.1. We define the design requirements of DSMC in Section 5.2. This is followed by a description of the system architecture, including the Dts-Overlay and how it interacts with DSMC in Section 5.3. As DSMC is influenced by the work done in CAR [20], we describe the existing CAR design in detail in Section 5.4. Finally, we describe in detail how DSMC is designed in Section 5.5 and we discuss some of the design choices made in Section 5.6.

5.1 Design Considerations

In this section, we describe key assumptions for the development of DSMC. We revisit the analysis findings of studying related work and discuss one related work which inspires the design of DSMC. For the development of DSMC, we rely on several key assumptions from the application scenario and the Dts-Overlay. We assume that:

- No pre-knowledge of which nodes serve as message carriers exists.
- The application scenario for our design is that of an emergency and rescue (ER) scenario.
- The node mobility in the ER scenario does not follow a random walk mobility model.

- There is a high chance of network partitions, because of the disruptive nature of the scenario.
- The nodes in the scenario are equipped with Wi-Fi capabilities and can together form a MANET.
- The MANET is sparse in nature due to the ER scenario.
- We can use the Dts-Overlay as previously described in Section 2.3.

To support DSMC in disruptive MANETs during an ER operation, we summarized and analyzed in Section 4.5, existing and related work. We found two general techniques used to support delay tolerant networks (DTN): (1) replication and (2) prediction. We concluded that using a prediction model to select carriers was the best solution. We also indicated that of all the work studied, CAR [20] offered the best ideas according to our initial requirements from Chapter 2.

The approach used in CAR to select carrier nodes is a prediction/probability based approach. The idea is to let each node estimate its probability of delivery to all other nodes and exchange these estimates with other nodes. This fits well with the Dts-Overlay since the overlay takes routing decisions in the overlay. If the overlay has up-to-date delivery probability tables from all other nodes, then it has a good foundation for smartly selecting the best carrier node for any destination. Another attractive feature of this approach is the possibility of letting different context attributes affect each delivery probability value. Context attributes could include anything measurable for a single node, like contact probability, available buffer space, remaining battery power, etc.

5.2 Design Requirements

We develop DSMC to provide a dynamic selection of message carriers in a disruptive MANET. The Dts-Overlay implements a delay tolerant overlay routing protocol for disruptive MANETs, but it assumes *a priori* knowledge about which nodes act as carrier nodes and require these to be configured before use. In this section, we list the requirements from the application scenario and the requirements of DSMC from a design viewpoint.

5.2.1 General Requirements

In the application domain we outlined requirements for an AV streaming service over heterogeneous, mobile and unstable networks in ER operations. We discussed the use of Dts-Overlay to meet these requirements. In this section, we revisit these requirements from the design viewpoint of the Dts-Overlay.

- **Delay tolerant:** The Dts-Overlay is built on top of the OLSR MANET protocol to provide delay tolerance. This is accomplished by implementing a SCF mechanism in the overlay, where nodes can store packets that have no route, carry packets and forward them when a route is up again.

- **Utilize non-random movement:** In the Dts-Overlay, there are predefined carrier nodes which move between network partitions. The overlay takes advantage of them to deliver packets between disconnected network partitions.

- **Best effort video streaming:** The Dts-Overlay attempts to minimize packets dropped by enabling MAC support [16]. It also attempts to minimize the delay by forwarding packets towards a destination, even when there are no connected paths to that destination. This is based on knowledge of previously existing next-hops.

- **Assume no infrastructure:** The Dts-Overlay assumes no existing infrastructure and instead relies on a MANET.

- **Resilient and fault tolerant:** The Dts-Overlay uses the MAC support functionality to avoid invalid entries in routing tables, which in turn leads to dropped packets. It also relies on its SCF mechanism to forward packets when routes between sender and receiver are disconnected.

- **Reactive to changes:** The Dts-Overlay forwards packets one hop at a time and lets the node at each hop utilize its own local routing information to affect the route taken.

- **Optimize resource usage:** As opposed to much of the related work, the Dts-Overlay does not use replication, which could potentially consume a big amount of resources.

- **No configuration:** The Dts-Overlay violates this requirement by requiring *a priori* knowledge about which nodes act as carrier nodes.

The violation of the "no configuration" requirement led us to one more requirement in the application domain.

- **Dynamic selection of message carriers:** Instead of relying on *a priori* configuration/knowledge, we should find a dynamic method of selecting the best carrier node for any destination node.

5.2.2 DSMC requirements

The requirement of selecting message carriers dynamically is the basis for DSMC. Here, we list the design requirements for DSMC:

- Use Dts-Overlay as a basis for covering the general requirements of an AV streaming service over a sparse MANET, such as in ER.
- Do not assume any *a priori* knowledge or configuration of nodes.
- Use probability of delivery to find the best carrier for any destination.
- Let each delivery probability value be a combination of any number of delivery probability (context) attributes.

5.3 System Architecture

DSMC is our design for a dynamic selection of message carriers. We are inspired by ideas from related work, e.g., CAR, and make use of composite delivery probabilities to identify the best message carriers to any destination. Local delivery probability sets are exchanged between nodes to obtain an up-to-date delivery probability table at each node. DSMC is designed to inter-operate with the Dts-Overlay, described Section 2.3, so that the Dts-Overlay can stop using predefined carriers. This does not mean that DSMC is dependent on the Dts-Overlay, as it could also be used with other overlay routing protocols that requires dynamic selection of carrier nodes.

5.3.1 Dts-Overlay Architecture

Figure 5.1 shows the main components of Dts-Overlay, flow of function calls and information exchange. The **Coordinator** component implements send and receive primitives for the video streaming applications. It forwards and receives packets from other nodes in the Dts-Overlay network. It uses the **Decision Maker** component to provide the next-hop IP address for a given packet destination. This address is obtained from the **Resource Manager**, which collects and monitors network state information. If no next-hop address can be found, the decision maker makes a call to the resource manager obtaining the closest carrier node (predefined carriers). Then it forwards the packet to this carrier node. If no carrier node is available, the decision maker looks up a cache of previously successful paths to that destination. The idea is that this previously successful path can be used to forward packets as close to arriving carriers as possible. If there are no entries in

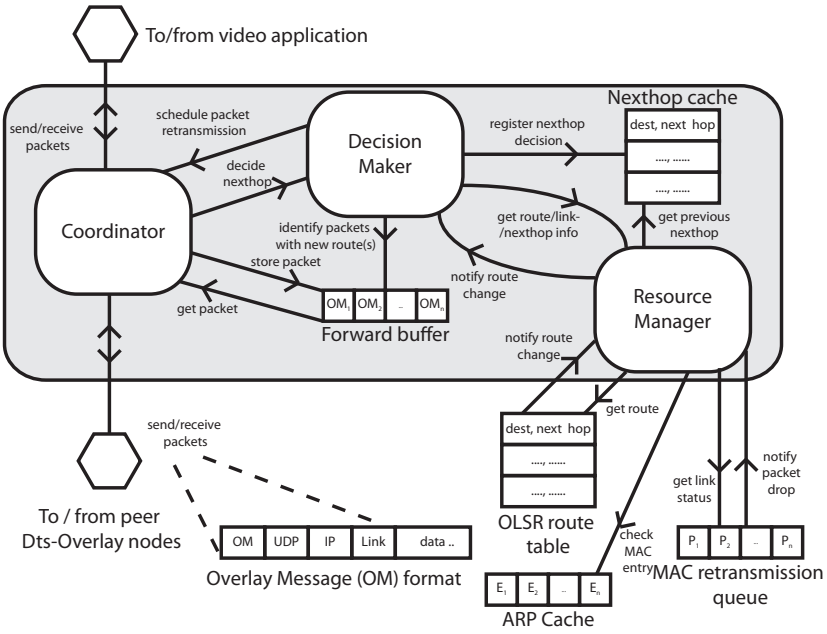

Figure 5.1: Dts-Overlay System Design (Lindeberg et al. 2011).

this cache, the sender stores the packet in the **Forward Buffer**, until a new route is discovered.

5.3.2 DSMC Interaction with Dts-Overlay

When we use DSMC with the Dts-Overlay (Figure 5.2), the decision maker no longer requests the closest carrier node from the resource manager. Instead, it forwards the request to a new component, i.e., the **Carrier Manager**. The carrier manager provides an interface for selecting the best carrier for any destination, i.e., it hides the complexity of DSMC from the Dts-Overlay.

The carrier manager is responsible for maintaining an up-to-date delivery probability table based on local and remote delivery probability estimates. When the request for a carrier comes from the decision maker, it looks up the destination in its local delivery probability table. There might not be an entry for that destination, because this node has not gotten any delivery probability updates for that destination yet. If this is the case, the decision maker buffers the packet in the **Forward Buffer**, otherwise the entry with the best carrier node is returned to the decision maker, which attempts to forward the packet to this carrier node.

A running thread initiates new calculations of delivery probabilities in the

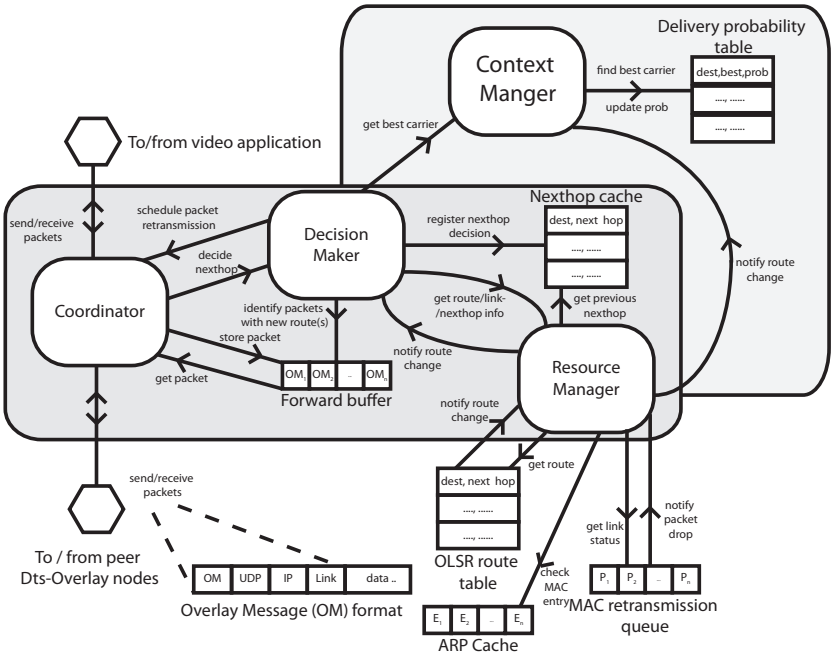

Figure 5.2: DSMC Design Combined With the Dts-Overlay System Design.

carrier manager at given intervals. The set of delivery probabilities is exchanged between nodes at regular intervals by piggybacking it on the underlying routing protocols link state update packets, such as OLSR HELLO messages or topology control (TC) messages. Upon reception of link state updates, the carrier manager extracts the delivery probabilities and updates its own local delivery probability table. Note that in this thesis, we simulate this behavior. This is described in detail in Section 6.2.5.

5.4 CAR Design

We discussed in Section 5.1 that we use ideas from related work when we design DSMC. In this section we describe, in detail, the design of Context-Aware Routing (CAR) protocol [20], which is an attempt at providing delay tolerant routing in MANETs. DSMC is heavily influenced by this work, and methods used in CAR are referenced later when the design of DSMC is described in detail.

The contribution of CAR is to use not only predictions of future node mobility, but to also include other factors, such as remaining battery power, which might affect a nodes delivery probability to another node. The CAR protocol is built on

top of the DSDV MANET routing protocol. DSDV is used to deliver messages between connected nodes in the same partition. If the destination node is not in the same partition as the sender, CAR attempts to find the best carrier for that message. The best carriers are defined as those that have the highest delivery probability. These delivery probabilities are synthesized locally from context information, such as the probability of nodes meeting. Delivery probabilities are exchanged between nodes by piggybacking it on the underlying routing protocols route updates. When nodes receive these updates, they extract the remote delivery probabilities, and update their local delivery probability table. If a node holds no information about a carrier candidate for a certain destination, it forwards the message to the node with the highest mobility. The idea is that the node with the highest mobility is most likely to meet the destination node.

5.4.1 Context Attributes

The context in CAR is defined as a set of attributes that describe the aspects of the system that can be used to improve the process of message delivery. This could be anything that can be measured locally at each node. In CAR, they use two context attributes:

1. The rate of change in connectivity.

2. Probability of being in the same partition as the destination node.

The first attribute is locally calculated by examining the percentage of a nodes neighbors that have had a change in their connectivity status between two instants, i.e., from connected to disconnected. The second attribute measures the amount of time those two nodes have been in reach of each other. It is estimated using a Kalman [14] filtering process, where the process is given the value 1 when the connection is up and the value 0 if the connection is down. The result of the process is an estimation of the probability of being in reach of the other node in the future.

They do not offer the details of their implementation of Kalman filters. However, in general, a Kalman filter is a fairly complicated algorithm with many variables. The big advantage of using it, is that it provides more realistic estimates of context attributes, even when there are no measurement available. Another advantage is that these estimates can be used to avoid exchanging tables between nodes more often than necessary, thus reducing the overhead of the protocol.

5.4.2 Combining Context Attributes

The context at one node is defined by a set of attributes $(X_1, X_2, ...X_n)$, where the X_i represent all possible values for that attribute, and x_i refers to a particular value within that set. All context attributes at one node are combined into one delivery probability value for each destination. The attributes are combined using a utility function U $(x_1, x_2, ...x_n)$. The goal is to maximize each attribute, i.e., to choose the node that presents the best trade-off between context attributes. This combined goal function can be defined as

$$Maximize\{f(U(x_i)) = \sum_{i=1}^{n} a_i(x_i) w_i U_i(x_i)\}$$

where w_i is the significance weight for each attribute. It reflects the relative importance of that attribute. For example, the second attribute in CAR is more significant than the first because regularly being in the same partition is a stronger indicator of delivery probability than a high mobility rate. $a_i(x_i)$ is the adaptive weight for each attribute, which itself is composite of three parts.

$$a_i(x_i) = a_{range_i}(x_i) * a_{predictability_i}(x_i) * a_{availability_i}(x_i)$$

The first is the criticality of a certain range of values, $a_{range_i}(x_i)$, e.g., low values of battery power should decrease delivery probability dramatically. The second is the predictability of the context information, $a_{predictability_i}(x_i)$. The last one is the availability of the attribute, $a_{availability_i}(x_i)$, i.e., not all attributes may be available for measurement.

The calculations of delivery probability values are done at regular time intervals at each node. Each node maintains a table of delivery probability values, which is updated at these time intervals.

5.4.3 Exchanging Delivery Probabilities

The calculated delivery probability values at one node are periodically sent to all nodes in that network partition, by piggybacking it on DSDV route updates. This might lead to some overhead, especially when there is no change in delivery probability values. The authors of CAR avoid some of this overhead by not sending the probability values if they have not changed. As they use Kalman filters for their context attribute values, they are able to use estimates when no updates are available. When nodes receive these route updates, they extract the new delivery probability table and they update their own local delivery probability table.

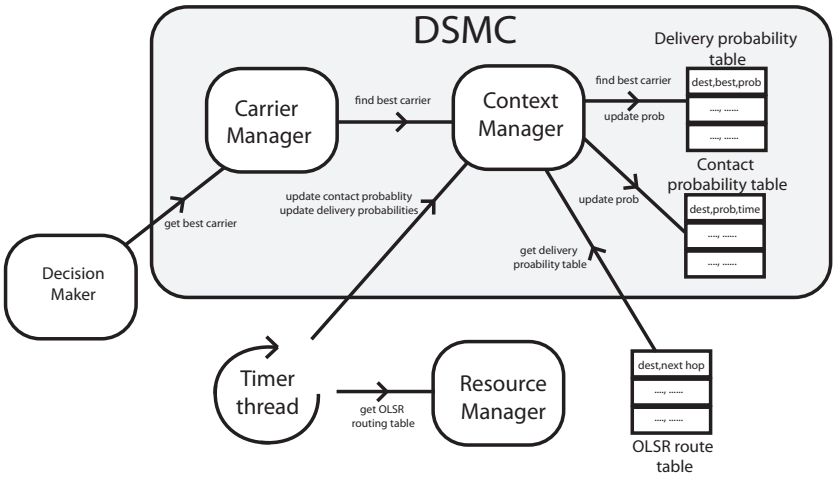

Figure 5.3: DSMC Detailed System Design.

5.5 DSCM Detailed Design

In this section, we discuss design issues and describe the design of DSMC in detail. An overview of the DSMC design, and its interaction with relevant parts of the Dts-Overlay, can be seen in Figure 5.3.

5.5.1 Carrier Manager

We start by describing the interaction between the **Decision Maker** and the **Carrier Manager**. When the decision maker has no route to a destination node, it makes a call to the carrier manager for the best carrier node for that destination. The carrier nodes responsibility is to hide different design strategies for finding the best carrier. This enables the separation of carrier selection from the decision makers routing logic. For DSMC, the carrier manager forwards the call for the best carrier node to the **Context Manager**.

5.5.2 Context Manager

The context managers responsibility is to find and store the best carrier nodes for different destinations. This is accomplished by maintaining a delivery probability table, where each entry contains the following:

- A destination node.
- The best carrier for that destination.

- The best carriers delivery probability value.

We maintain only one entry per destination, however, if we were to experiment with replication, we could maintain N entries per destination representing the N best carriers. For this thesis, we limit ourselves to not using replication and consider this as future work, see Section 8.3.

Delivery probabilities are estimated from a set of context attributes. As in CAR, these attributes describe the aspect of the system that can be used to improve the process of message delivery. In DSMC, we use only one attribute, which is almost the same as the second attribute used in CAR. They use the probability of being in the same partition, while we use the probability of direct contact.

Calculating the Contact Probability Attribute

To calculate the probability of one node being directly connected to another node, the developers of CAR use the Kalman filter process. In general, a Kalman filter is a fairly complicated algorithm with many variables. As mentioned earlier, the biggest advantage to Kalman filters is the possibility of good estimates even without available measurements. However, for the contact probability attribute, this measurement is always available. It will not always be completely up to date, due to the delay of route updates in the link state routing protocol, but it will be close. Therefore, we decided to use a more simple measurement technique for this attribute.

As in [10], we use an exponentially weighted moving average (EWMA) to calculate contact probability. A node i maintains a table of contact probabilities Eij for every other node j. Eij is updated each time instant, according to its current connections.

- If i meets j in this time instant: $Eij = (1 - a)[Eij]_{old} + a$
- If they do not meet: $Eij = (1 - a)[Eij]_{old}$

Where a is a constant parameter between 0 and 1. A low a value will result in less error in Eij, but also takes longer to achieve. A high a value will result in more errors but converge faster. This technique provides a simple and good estimate for contact probability. The contact probability attribute is measured, estimated and stored in the contact probability table in the context manager. The estimation of this attribute is initiated by the **Timer Thread** every second, which keeps this estimation up to date. The timer thread retrieves the OLSR routing table, stored in the **Resource Manager**, and forwards it to the context manager when it initiates the calculation of the contact probability attribute.

Combining Context Attributes

The context attributes at one node are combined into one delivery probability value for each destination. The challenge here is to combine them into one value that best estimates the actual delivery probability value. Even though we only use one attribute in this thesis, we design DSMC so that it can easily use more attributes.

We use a simplified version of the estimation of delivery probability values found in CAR. Specifically, we do not use the adaptive weight for each attribute. The reason is that while this weight is useful, it is not crucial for this estimation. Using the adaptive weight for attributes is instead listed as future work. Our estimation function is defined as

$$Maximize\{f(U(x_i)) = \sum_{i=1}^{n} w_i U_i(x_i)\}$$

where $w_1, w_2, ..., w_n$ are the significance weight of each attribute and $U_i(x_i)$ is the utility function representing the delivery probability to a specific destination.

The delivery probabilities are estimated in the context manager in time instants, e.g., each n seconds. This estimation is initiated from the timer thread, after each attribute has been measured. The delivery probabilities are then stored in the context managers delivery probability table.

If a node, regarded as a carrier node by others because of its delivery probability, stops moving between partitions, it should no longer be considered a carrier node. To achieve this, we implement the concept of aging delivery probability entries. Before delivery probabilities are estimated and compared to existing values, the existing values are aged by lowering them slightly. The speed at which such nodes are discovered is dependent on the aging factor, and when that node discovers a better carrier node. If we did not implement this aging factor, nodes that achieved a very high delivery probability would never be switched out, even if they stopped.

5.5.3 Exchanging Delivery Probability Tables

Using only locally estimated delivery probability values does not make sense, because they only represent that nodes delivery probabilities. In order to find the best delivery probability to a destination, nodes need to exchange these values. This can be accomplished by piggybacking this information on the underlying routing protocols link state updates. In DSMC, the underlying routing protocol is OLSR. This is due to the interaction with the Dts-Overlay, which uses OLSR.

The OLSR routing protocol makes a call to the context manager, via a cross layering mechanism, for its delivery probability table, for each HELLO message it sends out. When OLSR at any node receives a HELLO message, it extracts the remote delivery probability table, and send it to the context manager at that node. The context manager updates its local delivery probability table with the remote values, if they exceed existing ones.

The work in CAR has shown that it is possible to use the underlying proactive MANET routing protocol to exchange delivery probability tables. Due to the potential workload in implementing this idea, we use a simplified design model for exchanging delivery probability tables in this thesis. As we do not use the Kalman filter technique, we do not estimate delivery probability values and therefore we exchange more delivery probability tables. This means the overhead of this technique will be significantly larger than in CAR.

Our simplified exchange model works like this: Every four seconds, the NS3 simulator collects the delivery probability tables from each node and distributes them to every other node. However, due to the possible disruptive nature of the network, it is unlikely that all nodes are connected. So in order to be more realistic and mimic the method described above, tables are only exchanged between nodes that are listed to have contact in the OLSR topology. NS3 also provides us with this possibility, by implementing a cross layering protocol that allows access to the underlying OLSR routing protocol and its topology. The overhead of using OLSR to distribute delivery probability tables is estimated in Section 7.6. Time did not permit us to implement the exchange of delivery probability tables into OLSR HELLO messages and it is therefore left as future work.

The choice of simulating the exchange every four seconds is because this is a factor of two of what is used as the interval for HELLO messages in OLSR. The HELLO messages are used to update neighbors about a nodes connection. We do not think it necessary to exchange delivery probability quite that often and have settled for four seconds instead. As we aim to simulate the exchange using OLSR HELLO messages without introducing too much overhead, this time interval is a natural choice. A lower value would lead to more up-to-date values, however the overhead would also grow. A higher value would lead to less overhead, but less up-to-date values. This is obviously a trade off between overhead and how up to date our routing information will be.

5.5.4 Choosing the Most Optimal Carrier Node

In this thesis, we have only designed and implemented one context attribute. This attribute reflects the contact probability between two nodes. To understand why this attribute is not enough for DSMC to choose the most optimal carrier node,

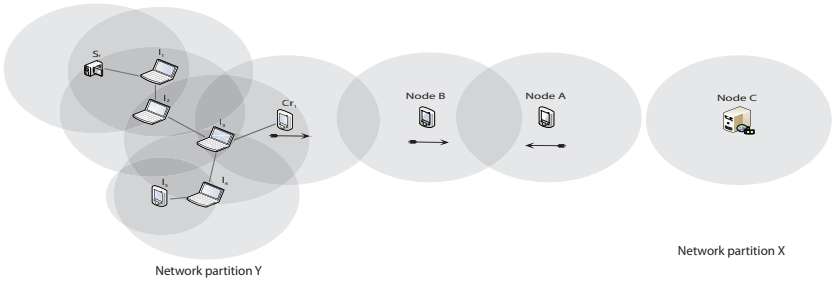

Figure 5.4: Routing Loop Example.

we consider a scenario situation.

There are two carrier nodes in the vicinity of a node holding packets for a destination. One of these carrier nodes is about to move towards the destination, while the other has just arrived from the destination. Which of these carrier nodes should/does DSMC choose?

As DSMC currently only uses the contact probability attribute, it will choose the newly arrived carrier node. This is because that carrier has more recently been in contact with the destination and therefore has higher delivery probability than the other carrier node. However, the most optimal choice of the two is the carrier that is about to leave, because then packets will arrive earlier.

As we can see from this example, the design of DSMC does not currently choose the most optimal carrier node in such a situation. One solution to this would be to design a new context attribute that reflected a carrier nodes probability of moving towards the destination. Time has not permitted us to investigate this further, and we instead list this as future work.

5.5.5 Routing Loops

One design challenge that has not been considered so far is routing loops. In a routing protocol where routing decisions are made hop-by-hop, there is a chance that packets will loop in the network. This is because any node making a routing decisions, has no idea where the packet has already been. In DSMC, we are particularly vulnerable to loops. This is due to the probabilistic routing approach. As an example, consider Figure 5.4 where node A is moving from network partition X to network partition Y. Another node B, is moving the opposite way. Node B is the carrier node for packets destined for node C in partition X. If node A and B meet while they move between the two partitions, they will exchange delivery probabilities. As node A just came from partition X, it will have a higher delivery probability value for node C than node B. This leads to node B updating its carrier

selection from itself to node A. Node A is moving the wrong way and brings the packets back to partition Y.

Carrier nodes could meet like this and so we cannot guarantee that there will not be loops between them without doing something to prevent it. To remedy this issue, we have considered several design suggestions:

- Avoid looping between carrier nodes by:
  1. Using IP addresses to avoid forwarding.
  2. Looking at carrier nodes delivery probability. If it is larger than a limit, do not forward any packets.
  3. Use clustering technique to discover that the carrier node is leaving a partition and that it should therefore not forward packets until it reaches the destination node.
  4. Introduce new context attribute(s) to avoid forwarding between carrier nodes.
- Avoid looping between all nodes by:
  - Use time to live(TTL) and/or split horizon with poison reverse mechanism.

The first four suggestions have in common that they attempt to avoid forwarding to other carrier nodes after they have been used as a carrier for that destination. The *first* suggestion is not optimal because it violates our requirement of "no configuration", i.e., it requires *a priori* knowledge about carrier nodes. Nodes would need to know which other nodes where carrier nodes by looking at their IP address, which is what DSMC is supposed to avoid.

The *second* suggestion is based on the fact that nodes that have met the destination node is likely still a carrier for them if their delivery probability exceeds some limit. In other words, carrier nodes could avoid forwarding as long as their local delivery probability exceeds this limit.

The *third* suggestion would require us to implement a clustering algorithm or use existing software to maintain knowledge of clusters and discover when a node is leaving a cluster. This in turn could be used to avoid forwarding by having nodes that discover they are leaving, not forward packets until they meet the destination node. One way to do this would be to use the igraph program [30] and topology information from OLSR to maintain cluster information at each node. The igraph program is a free software program for creating and manipulating undirected and directed graphs. One of it is features is to provide cluster information

from topology information. This is an interesting suggestion, however due to time restrictions, it is listed as future work.

The *fourth* suggestion is to design a new context attribute that would affect the delivery probability of nodes moving towards the destination. One idea, could be to use GPS to discover the direction of the moving node. However, if GPS is not available or unreliable, we are back to the same problem.

The *last* suggestion is a common technique used in many networking algorithms. TTL is a mechanism that limits the lifetime of a packet, i.e., it is a counter that is decremented for each node that handles it and that is dropped when the counter reaches zero. The TTL mechanism only prevents packets from looping more than a certain amount of times, it does not prevent looping. The split horizon mechanism omits routes learned from one neighbor in updates sent to that neighbor. Split horizon with poison reverse includes such routes in updates, but sets their metric to infinity. As we perform overlay routing, we would have to adapt this technique into the overlay. Exactly how this could be done is outside the scope of this work.

Time did not permit us to test out all of these techniques in this thesis. For simplicity, we use the *second* technique and look at a limit for delivery probability. Investigating the others and evaluating which is the best solution is considered future work.

5.6 Discussion

In this section we examine more closely some design decision made in DSMC.

DSMC is used together with the Dts-overlay in this thesis, but DSMC is designed so that it could work with other overlay routing protocols as well. DSMC does have some requirements of such and overlay, i.e., DSMC requires a cross layer technique to interact with the MANET routing protocol. This is due to the fact that DSMC needs to leverage routing information and to disseminate delivery probability tables.

We aim to use OLSR to read route updates and to disseminate the delivery probability tables in DSMC, but this does not mean that we necessarily require it. In theory, any proactive MANET routing protocol could be used to disseminate the probability tables. This means that context attributes which rely on OLSR route updates would have to be adapted as well. Using reactive MANET routing protocols would require a different approach as they do not disseminate information regularly.

In CAR, they use DSDV route updates to exchange delivery probabilities. The use of route updates, for both DSDV and OLSR, to exchange these delivery

probabilities might lead to large overhead, especially when there is no change in delivery probability values. CAR uses Kalman filters for context attribute values, which means it is able to use estimates when no updates are available. This means it can avoid a lot of overhead when there are no dramatic changes in delivery probability values. We do not use Kalman filters because of the complexity of implementing it, thus we accept higher overhead in our design. Implementing Kalman filters is instead listed as future work.

Chapter 6

Implementation

This chapter describes the implementation of DSMC into the NS3 simulator. First we introduce the implementation environment in Section 6.1. Then we look at the implementation of the different DSMC components in Section 6.2. Finally, in Section 6.3 we look at an overview of the whole system and describe how the different components work together with the Dts-Overlay.

6.1 Implementation Environment

We have chosen to use the network simulator 3 (NS3) [31], in this thesis because of the ease at which NS3 lets us implement our designs. In particular, NS3 offers smart pointers, trace sinks and object aggregation, i.e., NS3 models network nodes and protocols as objects and through smart pointers; thus we get easy access to these objects. Trace sinks allow us to get a signal when certain events occur in the simulation, such as a function being called. This allows us to intercept and output logging, for example when a packet is added to a buffer through a function call. These abilities make cross layering through function calls easy.

The Dts-Overlay was implemented in NS3 and because we use this overlay protocol to test DSMC, NS3 was also a natural simulator for us to implement DSMC. NS3 is a discrete-event network simulator used to, among others, simulate routing protocols, and is heavily used in wireless ad-hoc networking protocols research. NS3 is a clean slate implementation, but it is inspired by NS2, which is a widely used simulator in networking papers. It was released in 2008 and aims to replace NS2 due to some issues, including complex models, non-consistent results and some protocols models which are replete with bugs. NS3 is built using C++ and offers an implementation environment where developers can change existing networking protocols and implement new protocols. The core of NS3 supports

the use of many of the basic building blocks in a network protocol stack, such as IEEE 802.11, Csma, Wimax, UDP/TCP/IP, OLSR, AODV. All layers, from the physical layer to overlay and application layer are or can be implemented in the simulator.

6.2 DSMC Components

In this section we describe the actual implementation of DSMC in NS3 and the Dts-Overlay. We use Figure 5.3 as a reference throughout the description of the implementation. We start by looking at the Dts-Overlay components, as DSMC interacts with it. We then look at the main components of DSMC and we discuss how we implemented the exchange of delivery probabilities.

All classes can be found in the relative path *ns3.10/src/overlay/dts-overlay* in the provided code, if nothing else is specified, see Appendix B.

6.2.1 Dts-Overlay

All nodes in the simulation have the Dts-Overlay and DSMC installed. We use Figure 5.2 to describe the components of Dts-Overlay. The coordinator is implemented by the Dts-Overlay class and it is responsible for sending/receiving packets, both to the network and the video application. It communicates with the decision maker, to decide what to do with incoming packets. The decision maker interacts with the resource manager to find routes and with the carrier manager (DSMC) to find carrier nodes. The resource manager communicates with the MAC layer through a cross layer mechanism to get OLSR route updates and to check link status of connections.

6.2.2 Carrier Manager

The carrier manager class is responsible for identifying carrier nodes for destinations, regardless of the strategy implemented for finding them. This is accomplished by setting a variable *m_strategyType* for each simulation. Currently implemented strategies include the static carrier set-up used previously in Dts-Overlay and our DSMC mechanism. The reason for implementing this component is to easily switch and compare solutions without changing any code, i.e., control the strategy from command line arguments facilitated by NS3. The best carrier node for any destination, regardless of solution used, is provided using the function *GetBestCarrier(Ipv4Address destination)*. For DSMC, this function makes a call to the context manager for the best carrier node.

Table 6.1: Delivery Probability Table Entry

Field Name	Description	Example
m_destAddr	IP address of the destination	10.0.3.1
m_bestCarrierAddr	IP address of the best carrier node	10.0.2.1
m_bestCarrierDelProb	The best carriers delivery probability	0.89

6.2.3 Context Manager

The context manager is the main class of DSMC and is responsible for dynamically finding the best carrier node. This is accomplished by maintaining a delivery probability table of entries. Table 6.1 shows an example of such a populated table.

Delivery Probability Table

The table is called *r_deliveryProbabilities* and is implemented using a C++ map data structure:

$$map < Ipv4Address, DeliveryProbabilityEntry >$$

The DeliveryProbabilityEntry is one entry of the kind shown in Table 6.1. The use of map, which is basically a hash, is to provide fast lookup when using the function:

$$GetBestCarrier(Ipv4Addressdestination)$$

This is called by the carrier manager from the decision maker, when it needs to lookup the best carrier for a destination. The use of map is important because the lookup is done for every packet that has no routing path, which in a disruptive MANET could be quite often. It is also effective when we update the delivery probability table, as this also needs a lookup for each update entry. There is a small cost to iterating maps and adding entries to it, compared to other containers such as vectors. However this is not done as frequently and the cost is negligible compared to the benefit of fast lookup.

Updating the Delivery Probability Table

The delivery probability table is updated by the function:

$$UpdateDeliveryProbabilities(Ipv4Address\ dest,$$

$$Ipv4Address\ best\_carrier, double\ del\_prob)$$

This function updates entries if the *best_carrier* has a better delivery probability than the existing entry for that destination. If there exist no entry, then a new entry is added for that destination. This update function is called by the function *CalcDeliveryProbabilities()* which loops through all seen nodes and performs the calculation of delivery probability to each of them.

Context Attributes

Each delivery probability value is a combination of context attributes, as described in Section 5.5.2. In this thesis, we only use one attribute, the contact probability attribute. This attribute is calculated and maintained by the function:

$$UpdateContactProbabilities(vector < GenericRoutingTableEntry >,$$

$$r\_entries, int\ t\_slot)$$

This function is periodically called from the timer thread, which provides it with *r_entries*, which is the latest routing table obtained from OLSR. The function uses *r_entries* to update its own table of contact probabilities. This is stored in a map data structure:

$$map < Ipv4Address, ContactProbabilityEntry >$$

The contact probabilities are updated by looking at the incoming routing table to see which nodes are connectable neighbors and which nodes are not. The contact probabilities are calculated and updated as seen in Listing 6.1.

Listing 6.1: Pseudo Code for Updating Contact Probability

```
for each entry in OLSR_table
  contact = contacts.find(entry.destination())
  if contact and entry.GetDistance() == 1
    contact.probability = (1-alpha) * contact.probability
        + alpha
  else
    contact.probability = (1-alpha) * contact.probability

for each entry in contacts
  if contact != updated
    contact.probability = (1-alpha) * contact.probability
```

Nodes that are connectable neighbors get an increase in their contact probability and nodes that are not get a decrease. The last for-loop updates those

contacts that were not in the OLSR routing table. The alpha variable is a fixed constant where a low value gives a slow increase/decrease and a high value gives a fast increase/decrease. As this attribute reflects how probable it is that a node will have a connection to another node in the future, it should not decrease too fast because it needs to spread this information first. If the node is a carrier node, it needs to move between partitions before this information can be spread. Therefore, the alpha variable should relate to the speed at which carrier nodes move, the distance between partitions and the nodes transmission range. As we focus on an ER scenario in this thesis, we have modified the alpha variable to fit the scenario. However, for future work, we should work out a way to set this variable based on speed, distance and transmission range.

The delivery probability table is also updated by *AgeDeliveryProbabilities()* function. This function ensures that delivery probabilities decrease if they are not updated. Each time it is called, it decreases each delivery probability value with 5 %. The choice of 5 %, is based on preliminary testing of different values. This function is called periodically from the timer thread.

6.2.4 Timer Thread

The timer thread initiates function calls that need to be initiated periodically in the context manager. The NS3 simulator does not provide the possibility of using threads, so instead we schedule a function (*UpdateContext()*) to be run each second. This function first initiates the update of each context attribute. In this thesis, this is the contact probabilities through the function *UpdateContactProbabilities()*. Then it initiates the aging of existing delivery probabilities through the function *AgeDeliveryProbabilities()*. Finally, it runs the *CalcDeliveryProbabilities()* function and schedules itself to be run again each second.

6.2.5 Exchanging Delivery Probabilities

We have explained how delivery probabilities are calculated and stored. However, for DSMC to work, these probabilities need to be exchanged between nodes. As we indicated in the design of DSMC, we would like to use OLSR HELLO messages for this purpose. However, as we also indicated in the design, time did not allow us to implement this. Instead we use a simplified method. We utilize NS3 to exchange the delivery probability values. This is implemented in *cross-layer-exchange.cc* in the relative folder *ns3.10/scratch/dts/*, see Appendix B.

When we start a simulation, we also schedule a function, *ExchangeDeliveryProbabilities()*, to run every four seconds. This function is placed in the simulator in a way that allows it access to all node objects. We utilize this to set

up the exchange between each node. As we want this simulation of the exchange to mimic OLSR HELLO message piggybacking, we make sure those nodes cannot exchange delivery probabilities unless their OLSR routing tables list that they are connected neighbors.

Listing 6.2: Exchange of Delivery Probabilities

```
for nodeA in all_nodes
  for nodeB in all_nodes
    if nodeA != nodeB and nodeA.hasConnection(nodeB)
      nodeB.UpdatedDeliveryProbabilities(NodeA.
          GetDeliveryProbabilities())
```

The pseudo code for the exchange can be seen in Listing 6.2. We chose to exchange delivery probability tables every four seconds because this is similar to OLSR HELLO message interval. Although the OLSR HELLO message interval is two seconds, we double this because we want to minimize the overhead of exchanging tables and because initial test have shown that using two seconds does not improve delivery probability estimates significantly.

6.2.6 Avoiding Routing Loops

In DSMC, routing loops can occur between carrier nodes as explained in Section 5.5.5. To avoid looping between carriers, we implement a function in the context manager called:

$$CheckCouldBeCarrier(Ipv4Address\ addr, Ipv4Address\ dest)$$

This function checks the nodes delivery probability value towards the destination (*dest*), against a predefined limit value. The limit is set to 0.01 in this implementation. This value was chosen because nodes that have not met the destination node will have a delivery probability of 0. This works in scenarios where only carrier nodes connect disconnected network partitions. If some nodes visit both partitions once in a while, then they will avoid forwarding packets until their delivery probability value decreases, through aging, to a value below this limit. For DSMC to adapt to other scenarios, this limit should be investigated in future work. *CheckCouldBeCarrier* is called when a packet has no route to a destination. If it returns true, the packet is buffered until we have a connected path to the destination.

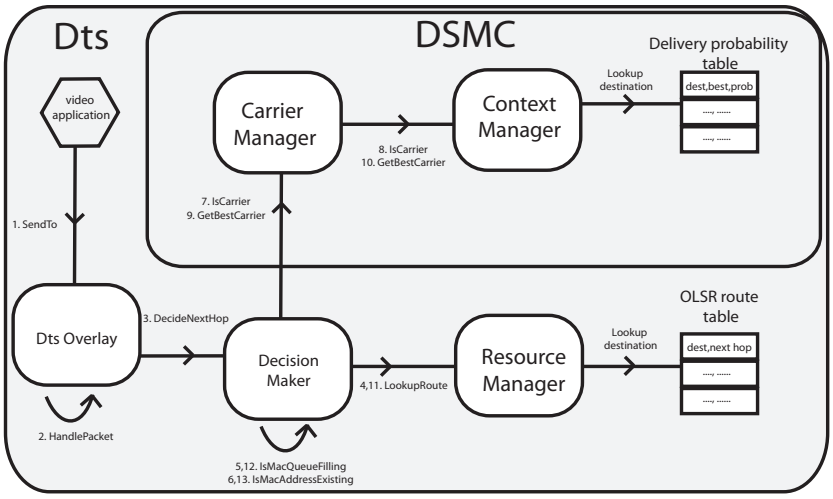

Figure 6.1: Dts-Overlay Interaction With DSMC Through Function Calls.

6.3 Dts-Overlay and DSMC Interaction

In this section, we give an overview of the DSMC components and how the Dts-Overlay interacts with DSMC in the process of routing packets.

To illustrate how DSMC works together with the Dts-Overlay in an ER scenario simulation, we follow one packets course through finding routing options for a destination. We use Figure 6.1, which shows the overview of the classes implemented, and the function calls made between them. The numbers in front of each function call is a reference to the sequence in which each function is called. These numbers can also be found in the following detailed description.

We start at the video application at the source node, which hands over the packet to the Dts-Overlay for delivery to a destination node. This is done by calling the function (1) *SendTo(packet p, Ipv4Address dest)* in the Dts-Overlay. This function adds an *OverlayMessage* header which contains the destination address for this packet. Then it calls the function (2) *HandlePacket(packet p)*, to decide what to do with the packet. HandlePacket makes a call to the function (3) *DecideNextHop(packet p)* in the decision maker class to find out where this packet should be routed.

In *DecideNextHop()*, we follow the pseudo code logic in Listing 6.3 to set the variable *DecisionType* for this packet at this node.

Listing 6.3: Pseudo code for Making a Routing Decision

```
if destination == local_node

  DecisionType = LOCAL_APP

else

  nextHop = LookupRoute(destination)
  if nextHop exist

    if !IsMacQueueFilling() and IsMacAddressExisting(
       nextHop)
      DecisionType = OTHER_NODE // Forward
    else
      DecisionType = LOCAL_BUFFER // Store

  else if IsCarrier(destination)
    DecisionType = LOCAL_BUFFER // Avoid carrier to carrier

  else

    carrier = GetBestCarrier(destination)
    nextHop = LookupRoute(carrier)

    if nextHop exist and !IsMacQueueFilling() and
       IsMacAddressExisting(nextHop)
      DecisionType = OTHER_NODE // Forward
    else
      DecisionType = LOCAL_BUFFER // Carry

return DecisionType and nextHop
```

First we check if the destination is the local node, in which case the packet has arrived at its destination. However, in this example, the packet is still at the source node and so we move on to check if OSLR reports a route to the destination by calling the function (4) *LookupRoute(Ipv4Address destination)* in the resource manager class. This function returns the next hop address for the destination according to OLSR if it is available.

If we find a next hop, we call the functions (5) *IsMacQueueFilling()* and (6) *IsMacAddressExising(nextHop)* to check the link status before we send the packet.

If the link status is not OK, then we buffer the packet by setting the *DecisionType* to local buffer. For the packet in this example, *LookupRoute()* reports no route because the source node and the destination node are in two separate network partitions.

The next step for *DecideNextHop()*, is to check if the local node is a carrier node for the destination. This is where our implementation of DSMC is executed. We make a function call (7) *IsCarrier(Ipv4Address destination)*, to the carrier manager. *IsCarrier()* checks which strategy is used for finding carrier nodes and makes a function call to (8) *IsCarrier(Ipv4Address destination)* in the context manager. The context manager *IsCarrier()* function looks up the node with the best delivery probability and compares its address to the local node. If they are one and the same, *IsCarrier()* returns true and the packet is buffered by setting the *DecisionType* accordingly.

The packet in our example is still at the source node which is not a carrier node. So we move on and call the function (9) *GetBestCarrier(destination)* in the carrier manager to find the best carrier node for that destination. The carrier manager makes a function call to (10) *GetBestCarrier(destination)* in the context manager. This function looks up the destination in its delivery probability table. If it has an entry for that destination, it returns that carrier nodes address to the decision maker. Now we know that we need to forward the packet to a carrier node for it to be delivered. We again use the stored OSLR routing information through the function (11) *LoookupRoute(carrier)* to find the next hop to that carrier node. If we find a route to that carrier node, we also check the link status through the functions (12) *IsMacQueueFilling()* and (13) *IsMacAddressExising(NextHop)*. If the link is OK, we use this next hop to forward our packet towards the carrier node. If the link is not OK, we buffer the packet. For the packet in our example, we have found a carrier node and a next hop address for that carrier node. We therefore set the *DecisionType* to another node and return from *DecideNextHop()*.

After we are done in the decision maker, we know what to do with our packet. If the decision was that the packet had arrived at its destination, then we deliver it to the waiting application. If the decision was that the packet should be buffered, then it is buffered until OLSR updates suggest that we have a routing path to the destination node or that a carrier for that destination is available. If the decision was that the packet has either a route directly to the destination or a route directly to a carrier node for that destination, then we forward the packet using the next hop address provided by the decision maker.

As the packet in our example arrive at different nodes, it is again handled by the Dts-Overlay which uses the decision maker as described to decided what to do with the incoming packet.

Chapter 7

Evaluation

This chapter presents our evaluation of the implementation of DSMC. The purpose of DSMC, is to provide a dynamic selection of carrier nodes in a disruptive MANET.

First, in Section 7.1 we describe our goals for this evaluation. In Section 7.2 we identify our methodology, and introduce the techniques and tools used. We present scenarios, scenario parameters and workload in Section 7.3. Section 7.4 evaluates the metrics used to analyze the performance of DSMC. The performance is studied in Section 7.5, and the overhead in Section 7.6. Finally, we summarize our main findings in Section 7.7.

7.1 Evaluation Goals

In order to evaluate the design and implementation of DSMC, we identify three goals:

- **Goal 1:** Verify that DSMC functions according to its design. This means that we need to test DSMCs ability to identify carrier nodes. For this, we run our algorithm over the ER scenario from [16]. Previously, the Dts-Overlay used static carriers. Now, we let DSMC dynamically identify carrier nodes.
- **Goal 2:** Evaluate the performance of DSMC with the use of performance metrics over different scenario configurations. We try to identify which scenario parameters affect performance the most by analyzing simulation runs in which Static-Dts and DSMC have a high variation in performance.
- **Goal 3:** Verify that the overhead of introducing DSMC scales well when the network grows. This is done using an analytical model for the overhead, where we look at estimates for average and worst case overheads.

7.2 Performance Analysis

There are three main approaches for performance analysis of computer systems. In this section we provide a short explanation of these techniques. We discuss our choice of performance analysis technique for two separate analyses, one for DSMCs performance, and another for DSMCs scalability.

1. **Analytical modeling:** Is the analytical model that captures the essence of a systems behavior, e.g., a networking protocols behavior. Then a mathematical method is used to analyze this model. This can effectively be used for small models with little effort. However, for large and accurate models, the complexity rapidly becomes high, e.g., a routing protocol in a challenging environments such as a disruptive MANET is very hard to model accurately because there are many variables that affect the system. To use analytical models, simplifications and assumptions can be made to lower the complexity. However, this lowers the accuracy of the results.

2. **Simulation:** Is to test the system in a simulation environment. The benefit is that the environment is modeled and acts predictably. This means that it is easier to isolate and find problems, as the exact same behavior and results can be achieved over two different simulation runs. One disadvantage of simulation is the difficulty of including every detail of a system in the simulation model.

3. **Measurements:** Is the real system with real life entities that is tested. This technique includes many parameters that is hard, if not impossible, to model in any simulation. For wireless networking, this includes weather, radio frequency interference and obstacles, contending services on any device running the software etc. However, there are several downsides. First of all, it is impractical and often difficult to implement and test code on devices, because devices are not always open for modification. Secondly, it is hard to monitor and collect metric statistics. These reasons also make it hard to deduce the actual reason for any result, good or bad. Finally, the results that are produced will be hard to reproduce by others. As an example it would be next to impossible to accurately reproduce the mobility of nodes in a MANET in two separate measurements.

In this thesis, we use the discrete-event simulator NS3, as described in 6.1, to implement and evaluate DSMC. NS3 allows us to set up a range of scenarios which can vary scenario parameters. As every layer of the protocol stack is simulated in NS3, we are able to monitor and gather statistics in detail, which would be hard if not impossible in any real life experiment. The Dts-Overlay was implemented

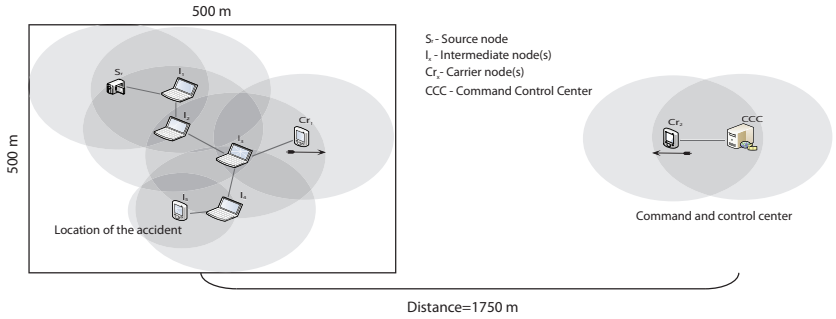

Figure 7.1: ER Scenario (Lindeberg et al. 2011)

in NS3 and the convenience of using that setup to fulfill *goal 1* and *goal 2*, is considerable. For the performance analysis of DSMC, we use statistics gathered from the NS3 simulator. For *Goal 3*, we analyze the overhead of DSMC using an analytical model. We do this because using an analytical model lets us easily see the effect on DSMC overhead when we scale up the system.

7.3 Scenario Setups

In this section we introduce the simulation environment and setup. We discuss scenario parameters and describe the two different scenarios used to evaluate the performance of DSMC.

7.3.1 Simulation Environment

We have conducted our main evaluation using the NS3 network simulator version 10, as presented in Section 6.1. Simulation runs are setup by providing scenario parameters as command line arguments to Dts, which is a simulation script in NS3 where the Dts-Overlay and DSMC is implemented.

A simulation run starts by configuring the need protocols and parameters. NS3 models each node and protocol with a separate object that we can configure. In this setup we configure each node in the simulation to use the IEEE 802.11b Wi-Fi standard in ad hoc mode. Each simulated devices for Wi-Fi uses the direct-sequence spread spectrum (DSSS) modulation and a constant data rate of 11 Mbps. We setup the Wi-Fi channel to model a constant speed propagation delay and the Friis propagation loss model is used to simulate signal loss. This setup is very common for MANET simulation studies.

Next we set up the nodes to use the OLSR routing protocol and the UDP

transport protocol. We also install the Dts-Overlay and DSMC at each node. Each node is assigned to an area in the scenario and given a mobility model for its movement, e.g., some nodes move randomly within a limited area and other nodes move only between such areas. Once we have set up the scenario (see Figure 7.1), we assign special mobility patterns and networking tasks to some nodes. We set up one node to be a client node which streams a video, to another node which serves a server node. We also set up some nodes to be carrier nodes, which move between the client network partition and the server (CCC) network partition. We end the simulation setup by scheduling the duration and then starting the simulation.

The simulations are run on a machine with the following parameters:

- CPU: 4 cores of 2.93 GHz each
- Memory: 8 GB
- Operating system: Linux, kernel 2.6.32-33-generic

The system parameters should not affect the simulation results. They only affect the time each simulation uses. For this reason, we do not consider the system parameters to be important for this evaluation.

7.3.2 Scenario Parameters

There are several parameters that are potential candidates for affecting the performance of simulation runs in a scenario.

- The **duration** of the scenario describes how many seconds the scenario should be simulated. Choosing extreme values of this parameter, such as only a few seconds will affect the results. However, such a choice is not a realistic value because it does not allow carrier nodes to move between network partitions. We chose to use 1800 seconds for all simulations, which should be more than enough time for DSMC and the Dts-Overlay to establish itself and start working.
- The **mobility** model used to model nodes movement is not varied. We use a random walk mobility model for nodes in the accident area, and a predefined mobility model for carrier nodes that move between the accident area and the CCC. The CCC mobility model is to stay still in one location. For future work, we would like to use scenario mobility traces from real ER scenarios.
- The **distance** between network partitions should have an impact on the performance of DSMC. A large distance between network partitions will cause a higher delay, because packets will have to be carried by carrier

nodes longer The delivery probability values will age more while the carrier nodes move between partitions. We therefore consider two scenarios, where the second one is scaled up in size and the distance is larger compared to the first one from [16].

- The **number of network partitions** could affect the performance of DSMC. Increasing the number of partitions increase the complexity of the scenario. This makes it harder to understand which variables and parameters affect the performance results and why. This applies to the **number of transmitting video source nodes** as well.
- Changing **number of carrier nodes** should affect results. This is because an increase/decrease in carrier nodes will lead to lower/higher delay as nodes need to wait for a carrier node to carry their packets. If there are enough carriers, then potentially the two partitions would merge into one. This since the carrier node swill then provide a continuous connection between two network partitions. We vary this parameter, to see the affects it has on performance.
- The **number of regular nodes** could affect the results, however not to the same degree as changing the number of carrier nodes. We do vary the number of nodes when we look at the scaled up ER scenario.
- The **speed at which nodes move** should not affect much for regular nodes when the speed is low. We do not consider the problems that come with high node speed in this thesis as nodes at an accident area are not likely to be moving any faster than walking speed (2 m/s). The speed for carrier nodes however should affect the delay and delivery probability values. Therefore we vary this parameter to see its affect on the performance.
- The **waiting time for carrier nodes** should affect delay and delivery probability values. We therefore vary this parameter to see its affect on the performance.

We have chosen two scenarios to study. Scenario 1 is the ER scenario used in the Dts-Overlay paper [16]. To make sure that DSMC also works in a different scenario, we scale up as mentioned some of the scenario parameters for Scenario 2.

7.3.3 ER Scenario 1

This scenario has a mobility model that aims to reflect mobility patterns from an ER scenario. For this we use the mobility model developed for the Dts-Overlay

[16], as shown in Figure 7.1. There are two network partitions in the ER scenario. Partition one; the CCC, and partition two; the area of the accident. A set of designated carrier nodes move with a varying speed between the two partitions, which are 1750 m apart. When carrier nodes arrive at either of the partitions, they stop and wait there for a varying amount of seconds before they return. Nodes in the accident area move with a speed of 2 m/s following the random waypoint mobility model [2] in a square area of 500 m x 500 m. There is one sending node in the accident area partition, and one receiving node in the CCC partition.

To evaluate how well DSMC works in this scenario for both *goal 1* and *goal 2*, we test the following identified parameters that we should vary in this scenario.

- **Number of carriers:** We vary the number of carriers to see how this affects the performance of the protocols. We vary between two, three and four carriers. We argue that one carrier only is of less interest in this evaluation, since DSMC and Dts would perform about the same. The performance of DSMC could suffer some initially because of the time needed to discover carrier nodes. We do not try more than four carriers because of the scenario size. If we used more than four carriers in our scenario, then the two network partitions would likely be connected by nodes, and would instead form one network partition. We expect that the number of carrier nodes will affect the amount of packets delivered, because the more carrier nodes there are, the more opportunities there are to utilizing carrier nodes for packet delivery.
- **Carrier speed:** We vary the speed of the carriers to see how this affects the efficiency of the protocol. The speed is varied from 2 m/s, to simulate walking speed for carrier nodes, and 10 m/s to simulate the carriers being vehicles of some sort. We expect the speed of the carrier nodes to affect the delay performance, because the speed affects how long a packet is buffered.
- **Carrier pause time:** We vary the pause time of carriers at the different network partitions to see how this affects the performance of the protocols. We try 30 seconds, 60 seconds and 120 seconds. We expect the pause time to affect delay, because the stop time affects how long a packet is buffered.

The different configurations are grouped by the number of carrier nodes and the performances of different configurations are explored further in Section 7.5.

7.3.4 ER Scenario 2

The second scenario is modeled after the first, but it is scaled up on some of the scenario parameters. The goal here is to see how well DSMC functions when the network and the scenario scales up. The parameters that we scale up are:

- **Distance:** We scale up the distance between the network partitions. From 1250 m to 2500 m. And we scale up the accident area from 500 X 500 to 1000 X 1000.
- **Number of nodes:** We increase the number of nodes in the accident area. From 12 to 24.
- **Number of carrier nodes:** We increase the number of carrier nodes moving between the accident area partition and the CCC. From 3 to 6.

7.3.5 Workload

Generally, workloads model the service requests to the system. For DSMC, the workload is a video stream that needs to be processed and split into packets before it is routed from one node in a network to another.

For both of our scenarios, we use a workload that consists of a single unicast stream of a video from a source node in one network partition (accident location) to a destination node in a different network partition (CCC). The video is 12 seconds long and it is repeated continuously throughout the scenario. The video resolution is 352x288 with a 25 fps frame rate. It is pre-encoded using H.264 standard, which represents a state-of-the-art in video streaming over networks. The target bit rate is 256 kb/s. We packetize it into an .mp4 container and encapsulated it into RTP-like packets, i.e., SecTs.

7.4 Metrics

Metrics are criteria in which we analyze the system performance. To evaluate DSMCs performance, we select the following five metrics:

- **Packet Delivery (PD):** Is the percentage of video packets delivered to the destination node.
- **Packet Loss (PL):** Is the percentage of video packets lost at any given time. This is from the destination nodes view point.
- **Packet Buffered (PB):** Is the percentage of video packets still stored at nodes.
- **Packet Delay:** Is the time from a video packet is sent from the source node until it is received at the destination node.
- **Overhead:** Number of extra bytes needed to exchange delivery probabilitiesper node per second.

PD is measured at the destination node by counting the amount of packets received. **PB** is measured by scheduling an event in the NS3 simulator that polls the amount of packets in each buffer at each node at the end of the simulation. **PL** is calculated as follows:

$$PL = PS - PD - PB$$

We measure packets sent (PS) from the sending application. All of these metrics are calculated as percentages of the packets sent. In the performance study, we also use the term PD_{Δ} which means the difference between Static-Dts and DSMC in **PD**.

The **packet delay** is measured per packet by looking at the SecTs header for each packet. This header is from the video application and contains a sequence number and a time-stamp from when the packet was sent. This time-stamp is extracted when the destination node receives the packet.

We estimate the **overhead** of using DSMC in terms of extra bytes per OLSR HELLO message. This is estimated using an analytical modeling, both for the average overhead and the worst case overhead. We study this later in a separate section (7.6).

7.5 Performance Study

In this section we study the performance of DSMC in our scenario simulations. Each scenario with a specific configuration is run five times with different seeds for the random number generator. The seed affects where nodes start (position) and which direction they move. This gives statistical variation to the scenario simulation runs, which we use to measure averages and standard deviations.

We evaluate DSMC against the static method of finding carrier nodes used in the Dts-Overlay, hereby only called Static-Dts. This is to see how using DSMC affects routing performance in the ER scenarios.

We vary between two, tree and four carrier nodes for scenario 1. For each of these, we vary the speed of carrier nodes between walking speed (2 m/s) and a vehicle speed (10 m/s). We also vary the amount of time a carrier spends waiting at each partition.

7.5.1 Configurations With Two Carriers

We start our performance study by analyzing the scenario configurations with two carrier nodes. Table 7.1 shows the different configurations of parameters.

Table 7.1: Scenario 1 Configurations with Two Carriers

Config	Carrier Speed	Carrier Pause Time
1	10 m/s	30 s
2	2 m/s	30 s
3	10 m/s	60 s
4	2 m/s	60 s
5	10 m/s	120 s
6	2 m/s	120 s

Each of these configurations has been simulated, and in Table 7.2 we list the average results of the five runs for each configuration, including the standard deviations (σ).

The first thing we observe from the results is that the PD_{Δ}, i.e., the difference in packet reception between Static-Dts and DSMC, are less than 2 % for configurations 1, 2, 4, 5 and 6. We also observe that for configurations 2 and 6, DSMC has slightly higher **PD** than Static-Dts. For Configuration 3, PD_{Δ} is 4.6 %, which is still a relatively small difference. Since PD_{Δ} is this low, we argue that the performance of DSMC and Static-Dts is close when there are two carrier nodes used in the scenario. DSMCs choice of carrier nodes is not always deterministic because it is based on a dynamical method that relies on several factors. One effect of this is the difference in standard deviations, where DSMC σ is about 3 times as high on average as Static-Dts. In all configurations, Static-Dts has less than 2 σ, while in DSMC all σ are below 6. We argue that the low standard deviation in the results suggest that both strategies have stable performance

If we look at how the parameters affect the results, we observe that configuration 2, 4 and 6 have **PD** between 70 and 80 % while configuration 1, 3 and 5 have **PD** between 80 and 95 %. From Table 7.1, we observe that configuration 2, 4 and 6 all use 2 m/s in carrier speed. This shows that lowering the speed of carrier nodes also lower the **PD**. The reason is that carrier nodes take longer time to move between partitions and therefore have fewer opportunities to deliver packets. If we look more closely at configuration 1, 3 and 5, which uses 10 m/s carrier speed, we observe that the **PD** decreases when the carrier stop time increases. Again, this is because carrier nodes have less time to move between network partitions.

The **PB** is very closely related to **PD** because the **PL** is close to zero for both strategies in all configurations. In other words, those packets that are not delivered

Table 7.2: Evaluation Results of Configurations with Two Carriers

Config	Strategy	PD	PB	PL
1	Static-Dts	95.23 % (σ : 0.23)	4.77 % (σ : 0.23)	0.01 % (σ : 0.00)
1	DSMC	93.87 % (σ : 0.62)	6.13 % (σ : 0.62)	0.00 % (σ : 0.00)
2	Static-Dts	70.08 % (σ : 1.62)	29.92 % (σ : 1.62)	0.00 % (σ : 0.00)
2	DSMC	71.49 % (σ : 5.31)	28.51 % (σ : 5.31)	-0.01 % (σ : 0.00)
3	Static-Dts	94.67 % (σ : 0.89)	5.32 % (σ : 0.89)	0.01 % (σ : 0.00)
3	DSMC	90.06 % (σ : 4.74)	9.94 % (σ : 4.74)	-0.01 % (σ : 0.00)
4	Static-Dts	72.11 % (σ : 0.87)	27.89 % (σ : 0.87)	0.00 % (σ : 0.00)
4	DSMC	71.54 % (σ : 3.70)	28.45 % (σ : 3.70)	0.00 % (σ : 0.00)
5	Static-Dts	85.78 % (σ : 0.35)	14.22 % (σ : 0.35)	0.01 % (σ : 0.00)
5	DSMC	84.09 % (σ : 1.17)	15.91 % (σ : 1.17)	-0.00 % (σ : 0.00)
6	Static-Dts	75.95 % (σ : 0.99)	24.05 % (σ : 0.99)	0.00 % (σ : 0.00)
6	DSMC	76.30 % (σ : 3.54)	23.70 % (σ : 3.54)	0.00 % (σ : 0.00)

are still in buffers at different nodes when the simulation ends. The reason for **PL** to be so close to zero, is due to Dts-Overlays MAC support functionality, as discussed more closely in Section 7.5.4.

From these results, we argue to prove that DSMC detects and utilizes carrier nodes to deliver packets, which was the *first goal* for this evaluation.

Configuration 6

There are some surprising results in Table 7.2. In two of the configurations, DSMC has a higher **PD** than Static-Dts. Though the difference is small (1.41 % and 0.35 %), we still expected Static-Dts to achieve higher **PD**, since carrier nodes are known. To investigate this difference, we include Figure 7.2, which shows **PD** for both Static-Dts and DSMC in one specific run for Configuration 6. Seed was for this run set to 5.

From this figure we make several observations. *First*, it takes a long time before either of the two strategies delivers any packets. For Static-Dts, it takes almost 600 seconds, and for DSMC it takes about 370 seconds. This is most likely due to the configuration setup, where there are two carrier nodes moving at only 2 m/s, and wait for 120 seconds, i.e., it takes a long time before carrier nodes reach each partition.

Second, we observe that DSMC delivers its first packets before Static-Dts.

The reason why this happens is that, for a brief time, the two carrier nodes are evenly spaced between the network partitions and they form a connected path between them. Static-Dts cannot utilize this because it avoids forwarding between carrier nodes. We also try to avoid forwarding between carrier nodes in DSMC to avoid looping, as described in Section 5.5.5. However, in DSMC we do not avoid forwarding when there is a connected path. Additionally, in the beginning of the simulation, carrier nodes in DSMC have not yet been able to classify themselves as carrier nodes. So there is nothing stopping a carrier node in DSMC to forward packets to another carrier node.

Third, we observe that the connected path disappears quickly and that nodes in DSMC must buffer packets until they can classify a carrier node. Once a carrier node with a high delivery probability arrives at the accident area, the delivery probability needs to be disseminated to the nodes at that partition. Once disseminated, nodes can utilize the arriving node as a carrier node. Nodes in Static-Dts do not need to wait for this as they know from the start which nodes act as carrier nodes. As a result, we observe that Static-Dts delivers more packets at around 600 seconds than DSMC does.

Fourth, we observe that at around 1300 seconds, DSMC has a spike in delivery, while Static-Dts does not. The reason for this is that a carrier node that is on its way to the CCC, gets a connection to the CCC through another carrier node. For a brief time, it is able to deliver packets through that other carrier node. When the other carrier node moves too far away from the CCC, the connection is lost. The first carrier node must wait until it gets a direct contact with the CCC node before it can send its remaining packets.

Last, we observe from the figure that, at the end of the run, DSMC delivery rate is higher than Static-Dts as the scenario is over. This means that DSMC has time to deliver more packets than Static-Dts. If the scenario had lasted longer, we believe they would have delivered the same amount of packets. This is because they have buffered the same amount of packets. This suggest that when they are finished delivering, they will have delivered the approximately same amount.

Delay analysis

The average delay for packets in all five runs for Configuration 6, is for DSMC 743 seconds and for Static-Dts 694 seconds. The average delay here is quite high; however the difference between DSMC and Static-Dts is low, e.g., only 49 seconds. We argue that the high delay here is due to the scenario parameters. In this configuration, the speed is 2 m/s and the wait time is 120 seconds.

We now study the combined CDF for all runs in this configuration. We use a CDF here to display the probability of different delay values for runs using this

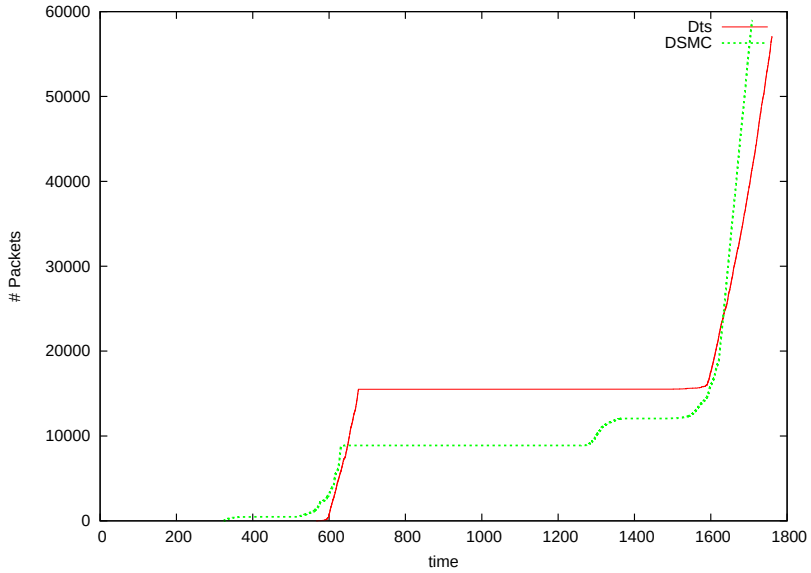

Figure 7.2: Single Run with 2 Carriers, Configuration 6

configuration. In this CDF, we observe that for $p <= 0.25$, DSMC has a better expected delay than Static-Dts. This is due to the connected path we discussed earlier, where DSMC could deliver packets through a connected path while Static-Dts could not. For $p > 0.25$, Static-Dts has a better expected delay than DSMC. We can understand the reason for this more clearly by again looking at Figure 7.2, where at 600 seconds, DSMC has delivered only about half of what Static-Dts has delivered. DSMC does not achieve the same level of delivery as Static-Dts before almost 1600 seconds has passed. We argue that this is what increases the expected delay in DSMC.

7.5.2 Configurations With Three Carriers

We now study the scenario configurations with three carrier nodes. Table 7.3, shows the different configurations of the scenario parameters. Configuration 9 is highlighted because that is the same configuration used in the Dts-Overlay paper [16].

Each of these configurations has been simulated, and in Table 7.4 we list the average results of the five runs for each configuration, including the standard deviations (σ).

From these results, we start by observing that PD_{Δ}, the difference in **PD** between Static-Dts and DSMC, is below 3 % for configurations 7, 9, 10 and 12.

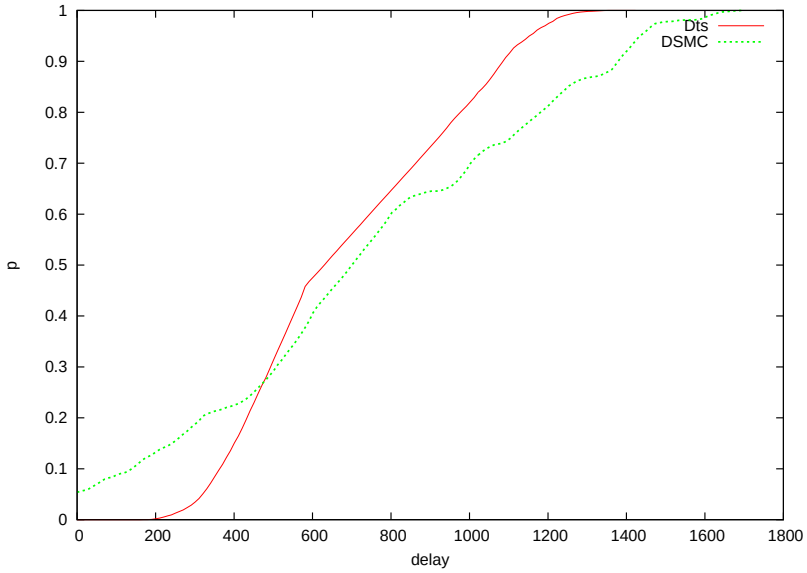

Figure 7.3: CDF for all Runs with Configuration 6

Table 7.3: Scenario 1 Configurations with Three Carriers

Config	**Carrier Speed**	**Carrier Pause Time**
7	10 m/s	30 s
8	2 m/s	30 s
9	10 m/s	60 s
10	2 m/s	60 s
11	10 m/s	120 s
12	2 m/s	120 s

Table 7.4: Evaluation Results of Configurations with Three Carriers

Config	Strategy	PD	PB	PL
7	Static-Dts	94.49 % (σ : 0.50)	5.51 % (σ : 0.50)	-0.00 % (σ : 0.00)
7	DSMC	92.36 % (σ : 0.57)	7.61 % (σ : 0.54)	0.07 % (σ : 0.00)
8	Static-Dts	44.60 % (σ : 1.26)	55.40 % (σ : 1.26)	-0.00 % (σ : 0.00)
8	DSMC	51.00 % (σ : 13.08)	49.01 % (σ : 13.08)	-0.01 % (σ : 0.00)
9	Static-Dts	88.97 % (σ : 0.51)	11.03 % (σ : 0.51)	-0.00 % (σ : 0.00)
9	DSMC	87.70 % (σ : 0.17)	12.29 % (σ : 0.17)	0.01 % (σ : 0.00)
10	Static-Dts	45.22 % (σ : 2.62)	54.78 % (σ : 2.62)	-0.00 % (σ : 0.00)
10	DSMC	44.28 % (σ : 11.72)	55.73 % (σ : 11.72)	-0.02 % (σ : 0.00)
11	Static-Dts	87.37 % (σ : 1.67)	12.63 % (σ : 1.67)	0.01 % (σ : 0.00)
11	DSMC	79.82 % (σ : 4.78)	20.17 % (σ : 4.78)	0.00 % (σ : 0.00)
12	Static-Dts	46.60 % (σ : 1.55)	53.38 % (σ : 1.57)	0.03 % (σ : 0.00)
12	DSMC	43.94 % (σ : 6.11)	56.06 % (σ : 6.11)	-0.00 % (σ : 0.00)

For Configurations 8, PD_{Δ} is 6.4 %. Here DSMC has the highest **PD**, and for Configuration 11, PD_{Δ} is 7.55 %. Even though PD_{Δ} is higher here than for scenarios using two carriers, we argue that PD_{Δ} is low enough to determine that DSMC performs nearly as well as Static-Dts when three carriers are used.

For configuration 8, 10 and 12 with 2 m/s movement speed, we observe that DSMCs standard deviation in **PD** is in average about 10, which is quite high compared to the standard deviations for two carrier nodes. The reason for this is again the unpredictability of carrier node choices. Additionally, we assume that if a connected path between the network partitions occurs, these last longer if the carrier nodes move slower. As these connected paths do not occur in every simulation run, we get high variations in the results. For configuration 7, 9 and 11, the standard deviation for DSMC is on average about 2, which suggest more stability when the speed increases. The standard deviation for Static-Dts is only about 1.4 on average, which is lower than for DSMC.

We now analyze how the scenario parameters affect **PD**. In configuration 8, 10 and 12, we observe a **PD** around only 50 % for both strategies, which is relatively low. This is again due to slow carrier speed. For configuration 7, 9 and 11, **PD** is between 80 and 95 % for both strategies. As with results from two carrier nodes, the increase in carrier stop time decreases **PD** for both strategies.

PB is very closely related to **PD**, again because the **PL** is close to zero for both strategies in all configurations. In other words, because almost no packets

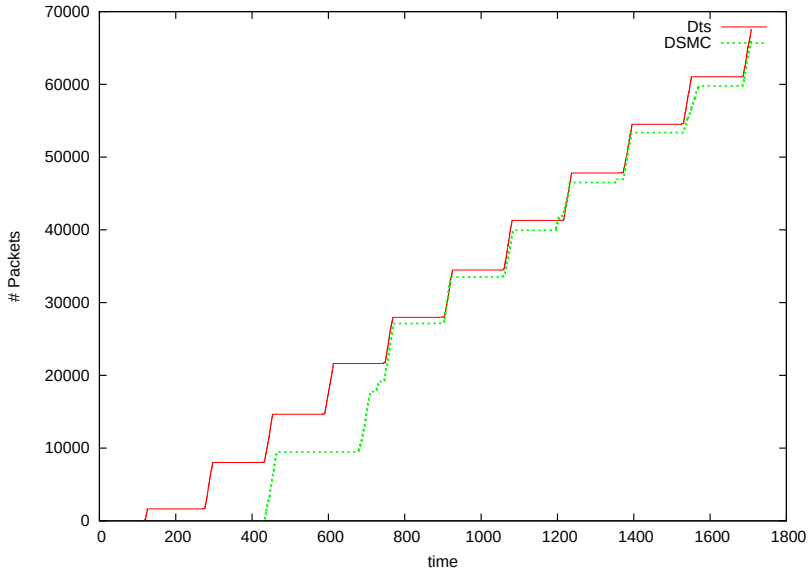

Figure 7.4: Run with 3 Carriers, Configuration 9

are lost, those packets that are not delivered are buffered. We analyze **PL** in more detail in 7.5.4.

Configuration 9

We study the configuration used in the Dts-Overlay paper [16], which is configuration 9, using three carriers with the speed of 10 m/s, and a waiting time of 60 seconds. From Table 7.4 we already know that **PD** for this configuration is very close for the two strategies ($PD_{\Delta} = 1.27$ %). In Figure 7.4, we show the **PD** of Static-Dts and DSMC in one specific run (with seed 5). We observe that it takes about 400 seconds for DSMC to deliver its first packet, while Static-Dts delivers its first packet after 100 seconds. This is due to the fact that nodes using DSMC needs some time to build up delivery probabilities and disseminate them before it can detect carrier nodes. While nodes wait for DSMC to detect carrier nodes, they buffer packets. Once DSMC can detect carrier nodes, the strategy starts to deliver the packets that were buffered. At around 800 seconds, Static-Dts and DSMC have almost the same **PD**. From there on to the end of the simulation, they deliver approximately the same amount of packets.

Configuration 8

We study a simulation run (seed 1) of Configuration 8 in Figure 7.5. DSMC surprisingly outperforms Static-Dts with a PD_{Δ} of 6.4 % for this configuration. There are three carrier nodes, with a speed of 2 m/s, and a waiting time of 30 seconds. We observe from the figure that the first packets in DSMC arrive before those in Static-Dts. The reason why, is that for a brief time, the three carrier nodes are evenly spaced between the network partitions, and they form a connected path between them. Static-Dts does not utilize this path, but DSMC does. This is the same event that occur for two carrier nodes in Configuration 6.

The connected path disappears quickly and nodes in DSMC must buffer packets until they can classify a carrier node. Nodes in Static-Dts do not need to wait for this, as they know from the start which nodes act as carrier nodes. As a result, we see that Static-Dts delivers more packets in the first 1000 seconds. After this, DSMC delivers more packets than Static-Dts. However, the reason for DSMC surpassing Static-Dts in the end is actually the simulation length. The trend for Static-Dts in this run is to deliver packets in a 600 second interval. If the simulation had lasted more than 1800 seconds, we would likely see Static-Dts and DSMC achieve close to the same **PD**. We observe that at around 1000 seconds and at around 1600 seconds, DSMC starts packet delivery before Static-Dts does. This is again due to forwarding between carrier nodes, when a connected path exists.

Configuration 11

We now study a simulation run (seed 4) of Configuration 11. The result of packet delivery is shown in Figure 7.6. We observe that while Static-Dts delivers its first packets at around 120 seconds, the first packets from DSMC arrive after about 530 seconds. This is again due to the time needed for delivery probability tables to build up and be disseminated. Additionally, carrier nodes wait for 120 seconds at each network partition. Once DSMC starts to deliver packets, it almost achieves the same **PD** as Static-Dts.

From around 530 seconds to 1300 seconds, DSMC and Static-Dts deliver about the same amount of packets, and we assume that they use the same carrier nodes. However, at about 1300 seconds, we see that DSMC deliver fewer packets than Static-Dts. This is most likely due to a non-optimal carrier node being selected in DSMC (See Section 5.5.4). From around 1500 seconds, it again starts to deliver buffered packets. From this we observe that DSMC sometimes chooses less optimal carrier nodes than Static-Dts. As carrier nodes wait 120 seconds at the accident, their delivery probability value towards the destination node is significantly reduce by the aging factor. It is therefore likely that an arriving carrier node has a much higher delivery probability than the carrier node that is waiting at the accident.

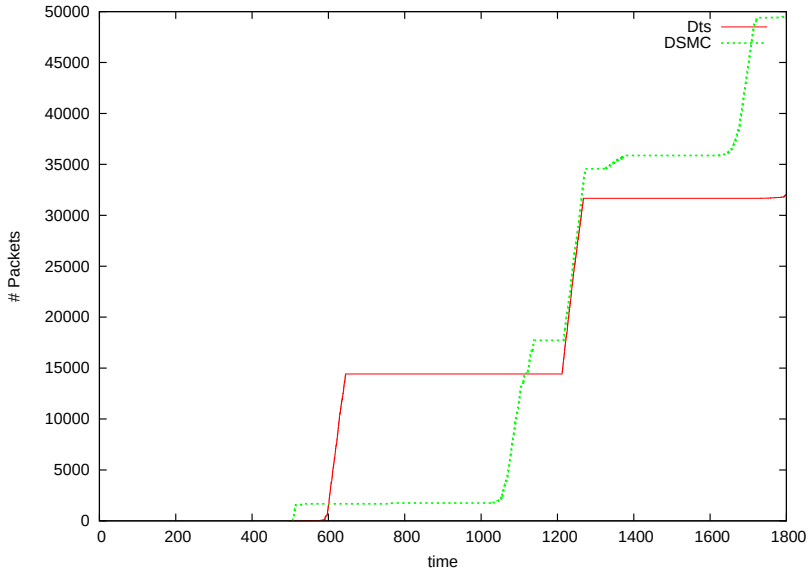

Figure 7.5: Single Run with 3 Carriers, Configuration 8

This observation is also backed by an increased delivery rate seen for DSMC at about 1700 seconds, which suggest that a carrier node has been buffering more packets than usual.

Delay Analysis

The delay for Configuration 8 is on average about 667 seconds for DSMC and about 604 seconds for Static-Dts. For Configuration 11, the average is about 350 seconds for DSMC and 210 seconds for Static-Dts. We observe that the difference in delay is low for Configuration 8 (63 seconds), and higher for configuration 11 (140 seconds). However, we do not go into to detail on these due to time restrictions.

We analyze the delay for Configuration 9 in detail, because this is the same configuration used in the Dts-Overlay paper. The average delay for all the packets in all five runs for Configuration 9 is for DSMC 256 seconds and for Static-Dts 163 seconds. If we look at Figure 7.4, we see that the first 10 000 (13 %) packets are not delivered until about 400 seconds. Additionally, for the next 10 000 packets, the delay is about 350 seconds. Although this figure only shows one of the runs that affect the average, we argue that initial lack of knowledge about which nodes are carrier nodes, is what affects the high delay the most.

We use a CDF to show and analyze the delay more closely, as for two carriers. The CDF for Configuration 9 can be seen in Figure 7.7. We observe that for both

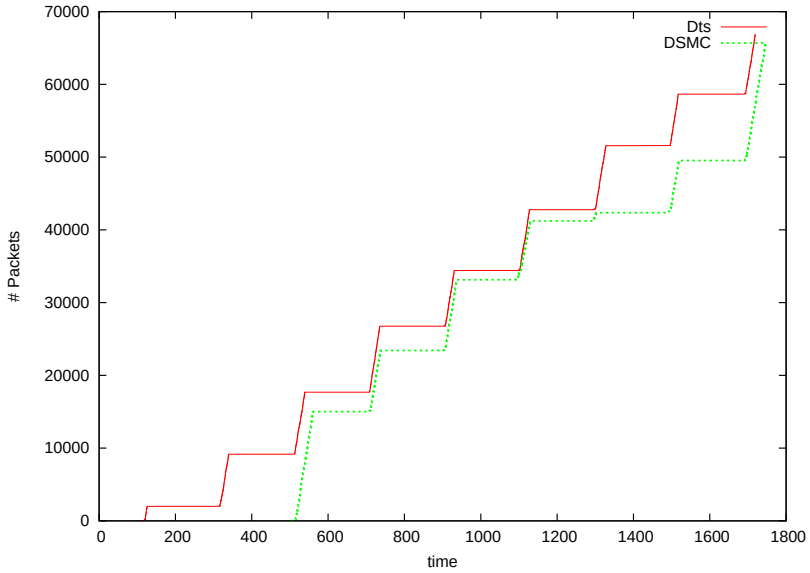

Figure 7.6: Single Run with 3 Carriers, Configuration 11

DSMC and Static-Dts, the lowest delay is about 70 seconds. This means that the probability of packets having a delay less than 70 seconds for this scenario configuration is close to zero. The expected delay for the two strategies are similar, though higher for DSMC, until about *p=0.7*. For $p > 0.7$, we see significantly higher expected delay for DSMC than for Static-Dts. We argue that this fits well with the observations we made concerning results discussed from Figure 7.4. As there were two intervals in that run where DSMC buffered packets, while Static-Dts delivered packets.

7.5.3 Configurations With Four Carriers

We now study the scenario configurations with four carrier nodes. Table 7.5 shows the different configurations of scenario parameters.

Each of these configurations has been simulated, and in Table 7.6 we list the average results from five runs for each configuration, including the standard deviations (σ).

The first thing we observe from the results is that PD_Δ, the difference in **PD** between Static-Dts and DSMC, is high (average 11.6 %). However, for configurations 14, 16 and 18, PD_Δ is negative, i.e., DSMC outperforms Static-Dts. These configurations are set up with 2 m/s carrier speed. We argue that this again shows that, when the carrier speed is low, connected paths between network partitions

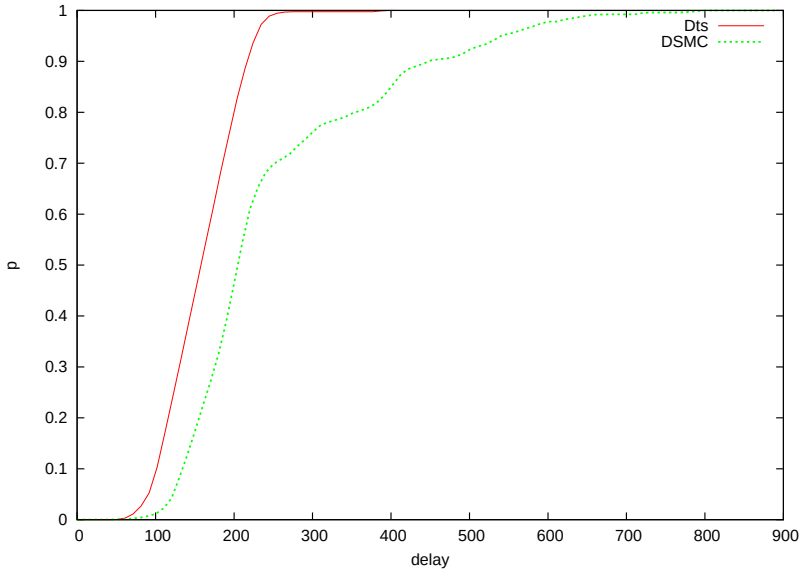

Figure 7.7: CDF for all Runs with Configuration 9

Table 7.5: Scenario 1 Configurations 2ith Four Carriers

Config	**Carrier Speed**	**Carrier Pause Time**
13	10 m/s	30 s
14	2 m/s	30 s
15	10 m/s	60 s
16	2 m/s	60 s
17	10 m/s	120 s
18	2 m/s	120 s

Table 7.6: Evaluation Results of Configurations 2ith Four Carriers

Config	Strategy	PD	PB	PL
13	Static-Dts	93.44 % (σ : 0.61)	6.56 % (σ : 0.61)	0.01 % (σ : 0.00)
13	DSMC	82.65 % (σ : 5.06)	17.34 % (σ : 5.06)	-0.00 % (σ : 0.00)
14	Static-Dts	56.58 % (σ : 1.13)	43.42 % (σ : 1.13)	0.00 % (σ : 0.00)
14	DSMC	69.15 % (σ : 7.24)	30.83 % (σ : 7.26)	0.04 % (σ : 0.00)
15	Static-Dts	90.86 % (σ : 1.35)	9.14 % (σ : 1.35)	0.01 % (σ : 0.00)
15	DSMC	82.04 % (σ : 4.20)	17.96 % (σ : 4.20)	0.01 % (σ : 0.00)
16	Static-Dts	56.89 % (σ : 1.05)	43.09 % (σ : 1.06)	0.06 % (σ : 0.00)
16	DSMC	69.25 % (σ : 7.42)	30.73 % (σ : 7.39)	0.06 % (σ : 0.00)
17	Static-Dts	88.45 % (σ : 0.41)	11.55 % (σ : 0.41)	0.00 % (σ : 0.00)
17	DSMC	70.90 % (σ : 2.62)	29.10 % (σ : 2.62)	0.02 % (σ : 0.00)
18	Static-Dts	59.46 % (σ : 1.39)	40.54 % (σ : 1.39)	0.00 % (σ : 0.00)
18	DSMC	66.95 % (σ : 1.92)	33.01 % (σ : 1.93)	0.18 % (σ : 0.00)

last longer. For configurations 13, 15 and 17, PD_{Δ} is positive, i.e., Static-Dts outperforms DSMC. In these configurations, connected paths last shorter due to the higher movement speed (10 m/s). We are not exactly sure why Static-Dts has such a high **PD** compared to DSMC. We suspect that, when there are four carrier nodes, DSMC has more carrier nodes to choose from. This means that DSMC is more likely to choose a non-optimal carrier node.

This is also reflected in the high standard deviations, which is on average 4.7 for DSMC and only 1.0 for Static-Dts. For three carrier nodes, we argued that that part of the reason for the high standard deviation is that connected paths do not occur in each simulation run. The standard deviation for DSMC is on average lower in this scenario than for three carrier nodes. We suspect that the standard deviation is on average lower for four carriers because the amount of carriers has increased. When the amount of carriers increases, we suspect that connected paths occur more often, and are more evenly spread among the simulation runs.

If we analyze how the scenario parameters affect the result, we observe some of the same behavior as for two and three carrier nodes. First, we observe that configurations with a carrier speed of 2 m/s have lower **PD** (55-70 %) than those with 10 m/s (80-95 %). This is again due to the time needed for carrier nodes to move between partitions, which lower their opportunities for delivery. Second, we observe that increasing the amount of stop time, decreases the **PD**. Both of these observations are made for both DSMC and Static-Dts.

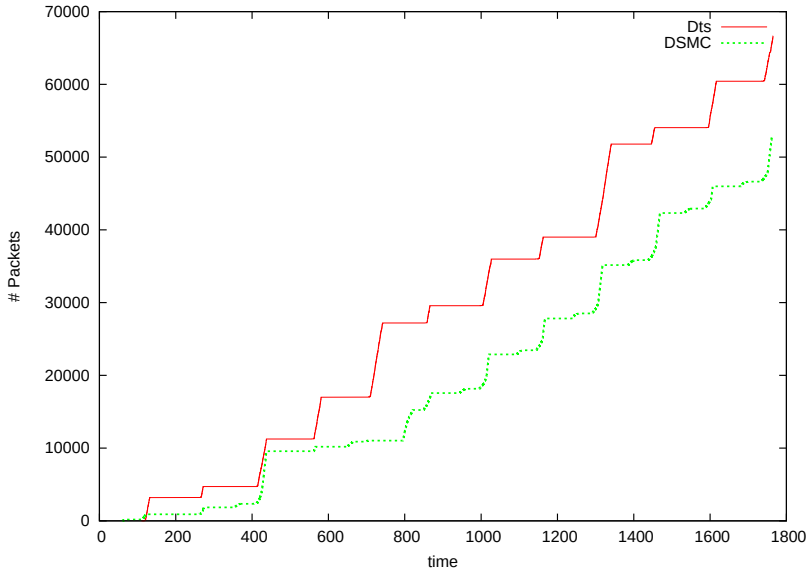

Figure 7.8: Single Run with 4 Carriers, Configuration 17

If we look at the **PL**, we again see that this is close to zero for all configurations for both runs. This is due to the MAC support of the Dts-Overlay and will be discussed more closely in 7.5.4. Due to the low **PL**, those packets that are not delivered are buffered (**PB**).

Configuration 17

We analyze one run (seed 2) of Configuration 17 in Figure 7.8, to examine further why Static-Dts has a **PD** of almost 18 % more than DSMC. In this configuration there are four carrier nodes with a speed of 10 m/s, and a waiting time of 120 seconds. We observe that DSMC delivers its first packets slightly ahead of Static-Dts. This is again due to a connected path between partitions. As there are four carriers, these provide a connected path from source to destination node at the start of this scenario run. The path that was there in the beginning, quickly goes away and Static-Dts achieves higher delivery than DSMC at about 100 seconds. DSMC is still waiting to identify carrier nodes and therefore buffer packets. We see that DSMC uses up to 800 seconds in this run before it stabilizes its delivery. Between 100 seconds and 400 seconds and between 400 seconds and 800 seconds, we observe that DSMC has almost no delivery of packets. This is again due to a non-optimal choice of carrier node, as described in Section 5.5.4, and the time need for delivery probability values to disseminate.

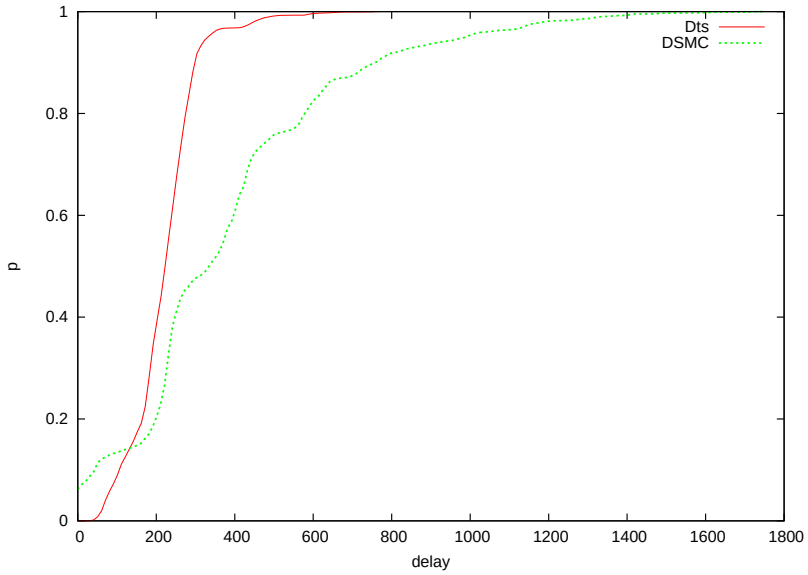

Figure 7.9: CDF for all Runs with Configuration 17

Delay Analysis

The average delay for all the packets in all five runs for Configuration 17 is 388 seconds for DSMC,and for Static-Dts, 225 seconds. The fact that the carrier nodes wait 120 seconds at each partition contributes partly to this delay. However, this is only about half the delay for Static-Dts and about a third of the delay for DSMC. Clearly, there is something else that affects the delay in this configuration. As we observed in Figure 7.8, there were two intervals where DSMC choose a non-optimal carrier. This led to almost no delivery of packets in those intervals. Those packets were instead buffered and we argue that this has increased the average delay considerably. To analyze this more closely, we examine the CDF for all runs for Configuration 17, in Figure 7.9. DSMC has better expected delay at $p <= 0.1$. This is due to connected paths being available for DSMC and not Static-Dts. For $p = 0.8$, the exposed delay for DSMC is almost 600 seconds, while for Static-Dts it is only about 250 seconds.

7.5.4 Packet Loss

From tables 7.2, 7.4 and 7.6, we observe **PL** close to zero for all configurations. This is due to the MAC support functionality of the Dts-Overlay, which both Static-Dts and DSMC uses. This leads to packets being buffered instead of being

dropped. As we have observed, some drop percentages are negative. 802.11 MAC uses positive acknowledgments. In case these are lost, MAC support will assume that also the original frame was lost, even though it might perfectly well have been received. This gives us duplicates, since MAC support will cause the packet to be retransmitted.

7.5.5 ER Scenario 1 Summary

We have studied the performance of DSMC using different scenario parameters. Through analysis and observations made from the performance results, we have verified that DSMC identifies and utilizes carrier nodes, as we sought to do for *Goal 1*.

In *Goal 2*, we aimed to evaluated the performance of DSMC against the performance of Dts-Overlay with static knowledge of carrier nodes and discover which parameters affect the performance the most. We have observed in our performance study that increasing the amount of carrier nodes increases the variation in performance between DSMC and Static-Dts. One reason for this is that nodes using DSMC sometimes chooses carrier nodes that are less optimal for delivery, when there is more than one carrier node in the vicinity, than those chosen by Static-Dts. As suggested in 5.5.4, choosing optimal carrier nodes for delivery is considered future work.

Another reason for the high variation is that DSMC can utilize connected paths between the two partitions, while Static-Dts can not. We observed that decreasing the carrier speed, lowered **PD**. In addition, we observed that decreasing the carrier wait time, increased **PD**. We argued that this is because the more opportunities carrier nodes have for delivery, the higher the **PD**.

When studying the delay, we have observed that DSMC generally has higher delay than Static-Dts. This is mainly due to the time needed for DSMC to disseminated and build accurate delivery probability tables. There is also an increase in delay when DSMC chooses a less optimal carrier node than Static-Dts.

7.5.6 ER Scenario 2

In this section, we analyze the performance of DSMC in ER scenario 2. Section 7.3.4 identified the scenario parameters that we scale up. We use Configuration 9 from Scenario 1 as a basis and scale up the following parameters: Distance from 1250 m to 2500 m, number of regular nodes from 12 to 24 and number of carrier nodes from 3 to 6. We have run this configuration five times and in Table 7.7, we present averages and standard deviations. Both of the configurations shown in this table make use of the same scenario parameters. However, there is a parameter

Table 7.7: Evaluation Results of Scenario 2

Config	Strategy	PD	PB	PL
1	Static-Dts	82.55 % (σ : 1.86)	17.45 % (σ : 1.86)	0.00 % (σ : 0.00)
1	DSMC	42.25 % (σ : 8.29)	57.73 % (σ : 8.28)	0.04 % (σ : 0.00)
2	Static-Dts	82.55 % (σ : 1.86)	17.45 % (σ : 1.86)	0.00 % (σ : 0.00)
2	DSMC	70.05 % (σ : 6.53)	29.97 % (σ : 6.54)	-0.05 % (σ : 0.00)

in DSMCs algorithms that is altered. We discuss this further after looking at the results from Configuration 1.

We observe that PD_{Δ} for Configuration 1 is 40.3 %. This suggest that DSMC does not work as well as Static-Dts when the size of the scenario area increases and when the number of nodes is increased.

In Figure 7.10, we show packets delivered for one run in scenario 2 (seed 1). This figure confirms the low average **PD**. Despite these initial results, we suspected that the reason for DSMCs low **PD** is not only due to the increase in number of regular nodes, carrier nodes and distance. As we explained in Section 6.2.3 of the implementation, DSMC relies on the static alpha variable. The alpha variable decides how quickly contact probabilities increase/decrease. When we scaled up the system, we did not alter this value. This means that nodes "forget" too quickly which nodes they have met, i.e., a carrier nodes contact probability to the CCC decreases to zero before it reaches the accident partition.

In Figure 7.11, we show a run (seed 1) from Scenario 2 where the alpha variable has been lowered from 0.02 to 0.005. We observe that, decreasing the alpha variable has indeed increased the performance of DSMC in this run. The average result on performance for this change is listed in Table 7.7 under Configuration 2. We observe that PD_{Δ} is only 7.57 %, which is much lower than Configuration 1, where the alpha variable was not lowered. This suggest that the alpha variable should not be static, as we already indicated in Section 6.2.3. Instead, it should somehow be more dynamic and reflect the speed at which carrier nodes move, the distance between partitions and the nodes transmission range. Investigating and improving this aspect is left as future work. We have not simulated configurations 1 to 18 from Scenario 1 with a lowered alpha variable. However, we suspect that it would not have affected results much. This is because the distance between the network partitions is lower, and therefore a lower alpha variable would not mean that nodes "forget" which nodes they have met.

As for performance results in scenario 1, the **PL** is close to zero due to Dts-Overlay MAC support functionality.

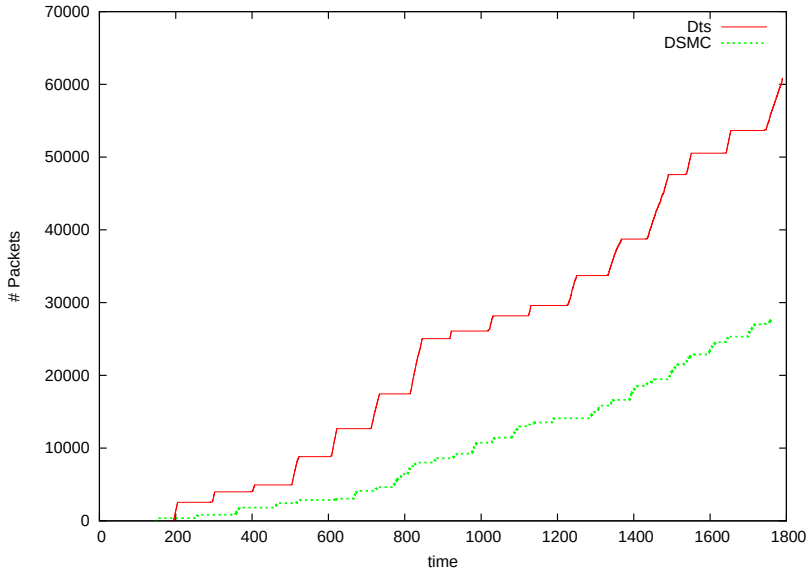

Figure 7.10: Scenario 2 Run with 5 Carriers

We do not go into the details of the delay for this scenario. We only present the average delays. The delay for Configuration 1 in Scenario 2 is on average 486 seconds for DSMC and 249 seconds for Static-Dts. For Configuration 2, the average delay is 346 seconds for DSMC and 249 seconds for Static-Dts. Static-Dts has the same delay for Configuration 1 and 2 because it is not affected by alpha variable. We observe that Configuration 2 has more than 100 seconds better average delay than Configuration 1 for DSMC.

7.6 Overhead Study

In this section we present an overhead study of DSMC. The overhead in DSMC comes from calculating, maintaining and exchanging delivery probability values between nodes. Exchanging delivery probability values is achieved by piggybacking it on OLSR HELLO messages, as explained in Section 5.5.3. We therefore estimated the overhead as the amount of extra bytes transferred per OLSR HELLO message. The overhead is evaluated in an analytical model. This is because there are only a few variables which affect this overhead, and because it easily lets us model the overhead when we scale up the number of nodes in the network.

The overhead should not only be measured in exchanged bytes. Calculating, maintaining and using delivery probabilities consume CPU cycles as well. However,

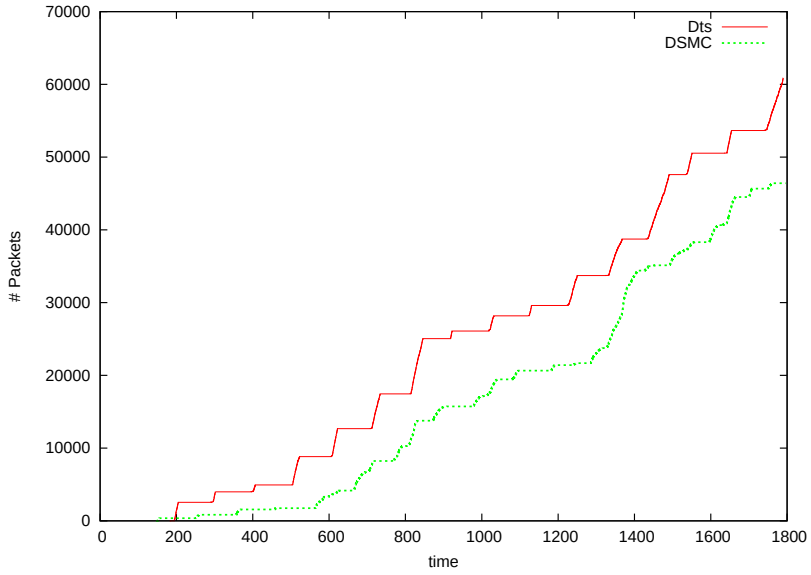

Figure 7.11: Scenario 2 Run with 5 Carriers, And a Lowered Alpha Variable

we lack an adequate model for estimating or measuring this overhead. Evaluating CPU overhead is outside the scope of this thesis and is instead left as future work.

There is also some overhead in the amount of extra storage that DSMC introduces, both in memory and persistent storage. However, this overhead is negligible compared to the amount of storage offered on mobile devices today.

Overhead Estimates

We use Configuration 9 in Table 7.3, as the setup for this overhead study. We estimate the overhead using two models. First, an average model which represents a sparse MANET. Second, a worst case model which represents a dense MANET.

We use the following model to estimate the average overhead ($\overline{O}$) in bytes per second, on each node:

$$\overline{O} = N * F * E * S$$

where N is the average number of connected neighbors, F is the exchange frequency, E is the average number of entries in a table and S is the size of each entry in bytes. We measure N and E over five runs in the ER scenario using Configuration 9 and we get this estimate:

$$\overline{O} = 5 * 0.25 * 6 * 12B = 90B/s$$

The exchange frequency, which is 4 Hz, and the size of each entry will not change in different scenario configurations. However, the number of connected neighbors and the number of entries in the table will increase if more nodes are added to the scenario. The question is: How will this overhead increase when we scale up the number of nodes in the MANET? As the table is piggybacked on OLSR HELLO messages, we compare this overhead to the size of these messages.

The format and use of OSLR HELLO messages is complicated, because it is used for link sensing, neighbor detection and MPR selection signaling. Additionally, there are rules for the size of one HELLO message and at which intervals it should be sent. We do not go in to the details of this complexity and so the following model is just an estimate. The OLSR HELLO message is packed into the data part of another packet, the general OLSR packet. The header for the general packet is 16 bytes. The HELLO message itself consists of 16 bytes worth of header for each interface at one device. One device can potentially have more than one interface, i.e., more than one radio transmitter. However, for this scenario and this thesis, we assume only one such interface per node. For each such interface, there is a list of neighbor interface addresses, where each address uses 4 bytes.

From this, we can make the following simplified model for the number of bytes exchanged in HELLO messages per second per node:

$$HELLO = (16B + 16B + (N * 4B)) * F * N$$

where N is the number of connected neighbors and F is the frequency with which HELLO messages are sent (every 2 seconds). If we use the same average numbers we used in the overhead estimate of DSMC, we get:

$$\overline{HELLO} = (16B + 16B + (5 * 4B)) * 0.50 * 5 = 130B/s$$

Using these estimates, we observe that DSMC increases the amount of bytes sent each second by about 66 %.

If we instead examine at the worst case overhead for this configuration, we use the same models, but exchange average numbers with worst case numbers. This means that we assume all nodes are connected to all other nodes. For configuration 9 with 16 nodes, this leads to the following estimation for the overhead of DSMC per node:

$$O_{worstcase} = 15 * 0.25 * 15 * 12B = 675B/s$$

and for HELLO messages per node:

$$HELLO_{worstcase} = (16B + 16B + (15 * 4B)) * 0.50 * 15 = 690B/s$$

The DSMC overhead in the worst case scenario is about the same as the OLSR HELLO messages. The difference of our estimate is only 15 B/s. To better observe

the relationship between the two, we show in Figure 7.12, the worst case overhead of DSMC against HELLO messages in bytes per second per node, as the amount of nodes in the MANET grows. It seems that from this figure, DSMC does not scale as well as HELLO messages. However, we should keep in mind that this worst case estimation is unlikely to occur, especially as the amount of nodes grows. If all nodes were connected to each other, we would have a dense MANET. In our ER scenario, we assume a disruptive MANET, where all nodes are not connected to all other nodes. The worst case scenario is not impossible and DSMC overhead has a problem with scalability when it occurs. However, this problem is not limited only to DSMC. OLSR overhead is known to suffer when the amount of nodes and density in the MANET grows, as discussed more closely in [22]. We have evaluated overhead for up to 200 nodes. For 200 nodes, DSMC in the worst case adds almost 150 % to the amount of bytes transferred per second per node.

In Figure 7.13, we compare the average overhead against HELLO messages as the amount of nodes in the MANET grow. This figure suggest that the overhead introduced by DSMC adds about 129 % to the amount of bytes used for transferring OLSR HELLO messages per second per node (for 200 nodes). The average numbers are just an estimation. We estimate that each node on average is connected to 1/3 of all other nodes and that each node holds delivery probability values and connection information for 4/9 of all other nodes.

Even though the difference in overhead between DSMC and OSLR HELLO messages does not increase much between the two models (21 %), the amount of B/s per node does. If we look at the difference in overhead for DSMC in the average and worst case models, we observe an increase of 660 % in number of B/s exchanged per node (for 200 nodes).

To summarize from this study, we have observed that the DSMC overhead more than doubles the overhead of OLSR HELLO messages, when the amount of nodes in the MANET increases. We have also observed that the overhead in DSMC and OSLR HELLO messages suffer when the density and amount of nodes is high.This is a known problem for OLSR [22]. However, as we target sparse MANETs in ER scenarios, we do not consider the worst case to be likely to occur.

7.7 Evaluation Summary

In this section, we summarize our evaluation results based on our initial goals from Section 7.1.

Our *first goal* was to verify that DSMC works. This was verified in Section 7.5.1 since PD_{Δ}, the difference in **PD** between DSMC and Static-Dts, was less than 5 % for configurations 1 to 6. This shows that DSMC detects and utilizes

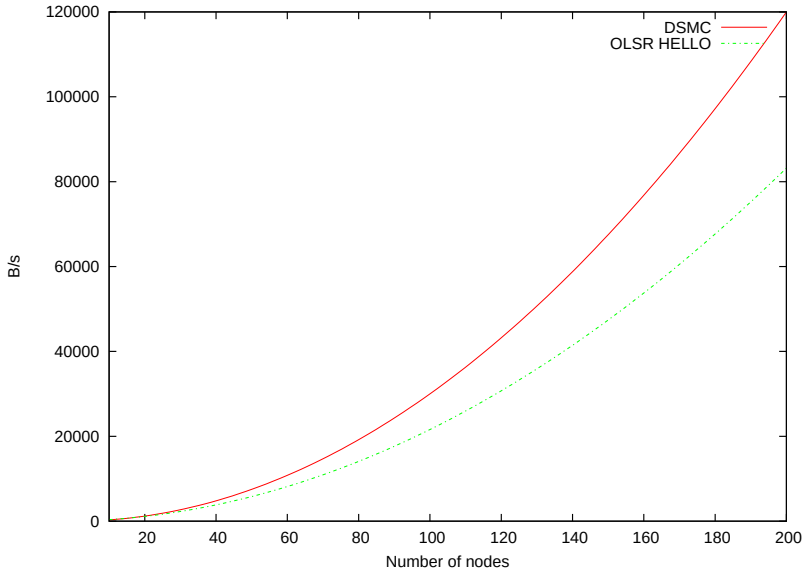

Figure 7.12: Worst Case Overhead of DSMC and OLSR HELLO Messages

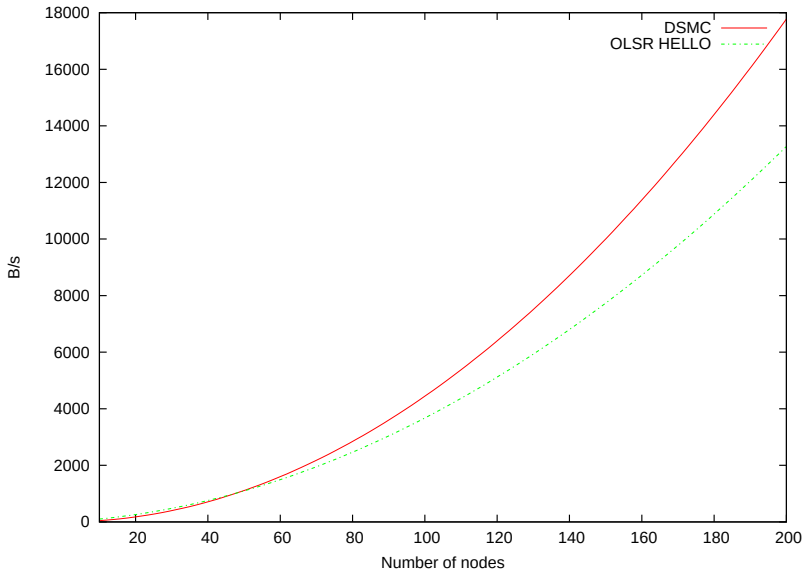

Figure 7.13: Average Overhead of DSMC and OLSR HELLO Messages

carrier nodes to deliver packets.

Based on our *second goal*, we examined different scenario parameters and how they affected the performance results. We have observed that of all the scenario parameters we varied; the number of carrier nodes has the biggest impact on PD_{Δ}. PD_{Δ} increased from two carriers to three carriers to four carriers. The speed at which carrier nodes move has the biggest affect on the average **PD**. When nodes move slowly, they have fewer opportunities to deliver packets. The stop time also affected the average **PD**. Long stop times also give fewer opportunities to deliver packets. Aside from the scenario parameters, we have observed that problems with the DSMC algorithm also affect the average **PD**. First, when DSMC chooses a non-optimal carrier node. Second, for Scenario 2, where the alpha variable needed to be adjusted to lower PD_{Δ} from 48 % to 7.6 %. From this we argue that the DSMC algorithm can be optimized and that the choice of alpha variable should be determined dynamically. One example left for future work is for alpha to reflect the speed at which carrier nodes move, the distance between partitions and the nodes transmission range.

When it comes to the delay, increasing the amount of time a carrier node stops and how fast it moves naturally increases the delay for both strategies. As expected, this is a dominant factor for delay. Beyond that, we have observed that nodes using DSMC need time to build up delivery probabilities. This has increased the delay of packets in DSMC. The fact that nodes using DSMC sometimes choose a less optimal carrier node than Static-Dts, also increases the delay. However, as the *a priori* knowledge required by Static-Dts is unrealistic, we argue that this delay is an acceptable trade off for a realistic solution.

Finally, we looked at the *third goal*, which was to verify that, the overhead of introducing DSMC scaled well when the amount of nodes in the network increased. We examined this in Section 7.6, where we modeled the average and worst case overhead of DSMC. The average model estimates the overhead in a sparse MANET, and the worst case model estimates a dense MANET. From our estimated average model, we have shown that DSMC scales well compared to OLSR HELLO messages when the amount of nodes in the network increases. However, for the worst case model, we have observed that both DSMC and OLSR suffers high overhead when the node density and number of nodes increases. We argue that the overhead scales well for an ER scenario, as we assume that the ER scenario is a sparse MANET, and not a dense MANET.

Chapter 8

Conclusion

In this master thesis, we have designed, implemented and evaluated a mechanism for dynamic selection of message carriers (DSMC) in a DTN for ER scenarios. In this chapter, we conclude our work. First, we present our contributions in Section 8.1. In Section 8.2, we make a critical assessment of this master thesis. Finally, Section 8.3 presents topics and ideas for future work.

8.1 Contributions

The goal for this thesis was to provide a mechanism for dynamic selection of message carriers. Our main contributions for this goal are: We analyze the application scenario and categorize related work. In this analysis, we argue that using a predictive approach to detect and utilize carriers fits best for a MANET in an ER scenario. Based upon our requirements analysis, we design a mechanism for detecting carrier nodes by having each node in the network calculate, maintain and exchange its delivery probabilities to other nodes. This design is implemented together with the Dts-Overlay, an ongoing development to tackle network disruptions through an overlay in the DT-Stream project [24]. We evaluate DSMC by comparing its performance against Static-Dts, which relies on *a priori* knowledge about which nodes act as carrier nodes. Through this evaluation, we show that DSMC functions as designed, that its performance is nearly as good as Static-Dts and that it induces higher delay. The overhead of DSMC is studied and we show that it scales well when the number of nodes in a sparse MANET increases.

Our *first* contribution is to analyze the application scenario and the Dts-Overlay system. Through the analysis, we identify key requirements from the DT-Stream project that Dts-Overlay meets and we identify one aspect of Dts-Overlay which does not hold up to the requirements. This is the *a priori* knowledge of

carrier nodes.

Our *second* contribution is to analyze and categorize related work in Chapter 4. We define two categories of DTN protocols: Replication-based and prediction-based. Through analysis, we identify arguments for why a prediction-based approach is better for our application domain. First, replication induces much overhead from the protocol. Second, most of the studied replication work did not utilize non random movement of nodes to improve delivery.

Our *third* contribution is to design a mechanism for dynamic selection of message carriers (DSMC). This mechanism calculates delivery probabilities at each node to all other nodes. We design our algorithm so that delivery probabilities are composed of any number of context attributes. The context attributes describe different aspects of the system that can be used to improve the process of message delivery, e.g., contact probability, node mobility and remaining battery. We made use of an attribute that represented the contact probability between two nodes. DSMC has been designed so that developers can easily implement any number of attributes, and add it to the calculation of the delivery probabilities. These delivery probabilities are exchanged between nodes.

Our *fourth* contribution is to implement DSMC in NS3. It has been implemented together with the existing Dts-Overlay. This was a challenge as it required us to understand how the Dts-Overlay worked and how we should implement DSMC as a separate component. DSMC is designed to be independent in that it could be used by any overlay to detect carrier nodes. This required us to extend the Dts-Overlay itself. However, DSMC does need access to route updates from OLSR or another proactive MANET routing protocol, to calculate delivery probabilities. It also requires the underlying MANET routing protocol to disseminate delivery probabilities regularly.

Our *fifth* contribution in this master thesis is the evaluation of DSMC. In this evaluation, we study the performance and overhead of DSMC in two ER scenarios. By observing that DSMC detects and utilizes carrier nodes, we verify that DSMC functions the way it was designed. For the performance study, we compare DSMC against Static-Dts, which is a strategy that assumes full knowledge about carrier nodes. In the performance study we show and argue that the performance of DSMC is nearly as good as Static-Dts.

We observe that, as we increase the number of carrier nodes in an ER scenario, the performance of DSMC and Static-Dts varies more. We identify two reasons for this: First, DSMC sometimes chooses a carrier node that is less optimal than Static-Dts. A less optimal carrier node is chosen more often as the number of carrier nodes increases. This is because, the more carrier nodes there are, the more likely it is that a sending node will have more than one carrier node in its vicinity. Second, DSMC can utilize connected paths between network partitions

through carrier nodes, while Static-Dts can not utilize connected paths. At high number of carrier nodes and slow carrier speed, these connected paths occur more often and last longer. If the MANET in the ER scenario changed from disconnected to connected, Static-Dts would not be able to deliver any packets, while DSMC would.

We also studied an ER scenario where we scaled up the size of the scenario area, and the number of nodes. In this study, we observed that DSMC performed poorly when compared to Static-Dts. We argue that this is due to a static variable in DSMC, which lowered contact probabilities. When the scenario is scaled up, contact probabilities are lowered too quickly, i.e., the carrier node contact probability to the CCC decreased to zero before it reached the accident partition. This leads us to the conclusion that this variable should be dynamic and reflect the speed at which carrier nodes move, the distance between partitions and the nodes transmission range.

DSMC induces only a limited increase in delay, due to DSMCs initial learning phase and its choice of non-optimal carrier nodes. Since the *a priori* knowledge required by Static-Dts is unrealistic, we argue that this increase in delay is an acceptable trade off for a realistic solution.

Through an overhead study, we estimated the overhead of DSMC. We estimated average and worst case overhead for DSMC against OLSR HELLO messages. We argue that the average model reflects a sparse MANET, while the worst case model reflects a dense MANET. We assumed a sparse MANET for an ER scenario and concluded that the overhead scales well when the amount of nodes in the MANET increases.

8.2 Critical Assessment

If we were to redo this master thesis, knowing what we do now, there are a few things we would consider doing differently. In this section, we briefly discuss these things.

We started off with a goal to find ways to detect and utilize carrier nodes automatically. In this initial period, I had other courses at the university which took most of my time. Some initial research and ideas were discussed. We spent some time implementing one of these ideas, based on the idea of broadcasting requests for carrier nodes when needed. Though we were able to implement it, the results were not promising, and the implementation brought more challenges than it solved. We decided then that we would do more thorough research into related work to see if we could get ideas for a better solution. A good deal of time was spent researching, analyzing and documenting related work. As I have had

a job on the side of my studies to support me financially, some time that should have been spent working on this thesis was spent working instead. As a result of this and the time spent on the initial ideas, there was only about 6 months left until the thesis was due (including summer vacation), when we finally decided on what to do. This time was spent designing, implementing, evaluating DSMC and writing this thesis. I wanted to finish the thesis within normal time, and so many of the following things would have been possible if I instead had got an extension.

In hindsight, we should have started the thorough study of related work sooner and I should have cut back on working on the side. If we had more time, we would have liked to do some things differently. We should have spent more time designing and developing DSMC. This way, we could for instance have better handled optimal choice of carrier nodes, which is a problem for DSMC. Initially, we only considered finding possible carrier nodes, and not the most optimal one.

We would also have spent more time making the alpha variable dynamic, as the performance depends heavily on it being adjusted to the scenario parameters.

As we have mentioned in Section 5.5.3, we simulate the exchange of delivery probabilities. Initially, we had hoped that we would have had time to implement this using OLSR hello messages. However, we decided that we would first implement this by simulation, and then focus on the rest of the DSMC implementation. When the basic implementation was finished, we realized that we would not have time to implement this exchange in this thesis.

We would also have spent more time developing and testing different context attributes. Some of the ideas we have had are described in the following future work section.

For the evaluation, we should have tested Dts-Overlay with DSMC against a standard protocol for DTN, such as epidemic routing. As we argued in related work that replication based protocols induced much overhead, it would have been good to compare the overhead between the two. However, there was no such protocol already implemented for NS3, and we did not have the time to implement it ourselves.

In a thesis like this, we discover many things that would be interesting to investigate further. Even though there were several things that we did not have time to study and investigate, I am quite satisfied with my work.

8.3 Future work

Through the course of working with this master thesis, we have discovered several topics and ideas that might be worth investigating. Due to time restrictions, we have not had time to explore them further.

- As we mentioned in Section 5.5.4, the current design of DSMC does not try to choose the most optimal carrier node when several possible carriers exist in the vicinity. In our evaluation of DSMC, we have seen that choosing a non-optimal carrier node decreases the performance of DSMC. For future work with DSMC, we would like to investigate ways to affect this choice. One way would be to implement a new context attribute, which reflects a carrier nodes likelihood of leaving.

- Implement the use of OLSR HELLO messages to exchange delivery probability values: This will remove the need to rely on NS3 to exchange these values. This would also make DSMC more realistic, i.e., closer to a version that can be used on a real device (see Section 5.5.3).

- Extend the combination of context attributes to include adaptive weights: Using adaptive weights would allow us to have more control over how each context attribute affects the delivery probability (see Section 5.5.2).

- As indicated in the implementation and in the evaluation, the choice of the alpha variable value is essential for performance. In the current implementation, this value is set as a static value. Instead, it should be set dynamically according to the distance between partitions, node speed and node transmission range. For most ER scenarios, we could assume that transmission range and node speed are predictable, and in that case the alpha variable would only have to reflect the distance between partitions.

- Design and implement more context attributes:

  - Use GPS to capture movement towards/from destination.
  - Use the trend of a buffer, i.e., see if the trend of a node is to empty its buffer or fill it up. A node has a buffer for every destination.
  - Use an energy model to help prevent one carrier node from handling more packets than other carrier nodes, and thus consume too much battery.
  - We have observed that DSMC often requires a long time before it discovers carrier nodes. This increases delay as it forces nodes to buffer their packets. One solution to this could be to introduce a new context attribute which measures the mobility of nodes, i.e., measure how fast a node moves.

- It would be interesting to experiment with replication. This could be done by storing the N best probability values and their addresses. This would

allow DSMC to make use of N carrier nodes to improve performance at the cost of an increase in overhead.

- In the related work of CAR [20], Kalman filters were used to decrease the amount of delivery probability exchanges. If we were to implement Kalman filters in DSMC, it would greatly reduce the overhead, especially between nodes which stay connected/disconnected for long time periods, because the Kalman filters would be able to predict delivery probability values from other nodes.

- Routing loops can occur between carrier nodes because routing in DSMC depends on delivery probabilities. As an example, consider a carrier moving towards a destination when it meets a carrier that has just been in contact with that destination. The first carrier will forward its packets to the other carrier, because it has a better delivery probability. As discussed in Section 5.5.5, there are several different options available to solve this. It would be interesting to investigate these options further.

- As of November 2011, there is a new version of NS3, version 12. For future work and further use of DSMC, it would be a good idea to update DSMC to use this new version. Since newer versions usually have fewer bugs and because others that might want to use DSMC in their work is probably developing in newer versions of NS3.

- We would like to test DSMC in more scenarios, e.g., real traces from emergency operations. This would perhaps reveal other weaknesses that we have not considered in this thesis.

Bibliography

[1] Ameer Ahmed Abbasi and Mohamed Younis. A survey on clustering algorithms for wireless sensor networks. *Computer Communications*, 30(14-15):2826 – 2841, 2007. Network Coverage and Routing Schemes for Wireless Sensor Networks.

[2] Christian Bettstetter, Giovanni Resta, and Paolo Santi. The node distribution of the random waypoint mobility model for wireless ad hoc networks. *IEEE Transactions on Mobile Computing*, 2:257–269, 2003.

[3] John Burgess, Brian Gallagher, David Jensen, and Brian Neil Levine. Maxprop: Routing for vehicle-based disruption-tolerant networks. In *In Proceedings IEEE INFOCOM*, 2006.

[4] B. Burns, O. Brock, and B.N. Levine. Mv routing and capacity building in disruption tolerant networks. In *INFOCOM 2005. 24th Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings IEEE*, volume 1, pages 398 – 408 vol. 1, 2005.

[5] Sergio Cabrero, Paneda Xabiel, Thomas Peter Plagemann, Vera Hermine Goebel, and Matti Siekkinen. Overlay solution for multimedia data over sparse manets. *Proceedings of the 2009 International Conference on Wireless Communications and Mobile Computing: Connecting the World Wirelessly*, 2009.

[6] Ionut Cardei, Cong Liu, Jie Wu, and Quan Yuan. Dtn routing with probabilistic trajectory prediction. In *Proceedings of the Third International Conference on Wireless Algorithms, Systems, and Applications*, WASA '08, pages 40–51, Berlin, Heidelberg, 2008. Springer-Verlag.

[7] Chao Chen and Zesheng Chen. Evaluating contacts for routing in highly partitioned mobile networks. In *Proceedings of the 1st international MobiSys workshop on Mobile Opportunistic Networking*, MobiOpp '07, pages 17–24, New York, NY, USA, 2007.

[8] Yang Chen, Wenrui Zhao, Mostafa Ammar, and Ellen Zegura. Hybrid routing in clustered dtns with message ferrying. In *Proceedings of the 1st international MobiSys workshop on Mobile opportunistic networking*, MobiOpp '07, pages 75–82, New York, NY, USA, 2007. ACM.

[9] T. Clausen and P. Jacquet. RFC 3626: Optimized Link State Routing Protocol (OLSR). RFC 3626 (Experimental), October 2003.

[10] Ha Dang and Hongyi Wu. Clustering and cluster-based routing protocol for delay-tolerant mobile networks. *IEEE Transactions on Wireless Communications*, 9(6):1874 –1881, June 2010.

[11] A. Elwhishi and Pin-Han Ho. Sarp - a novel multi-copy routing protocol for intermittently connected mobile networks. In *Global Telecommunications Conference, 2009. GLOBECOM 2009. IEEE*, pages 1 –7, 30 2009-dec. 4 2009.

[12] D. Johnson, Y. Hu, and D. Maltz. RFC 4728: The Dynamic Source Routing Protocol (DSR) for Mobile Ad Hoc Networks for IPv4. RFC 4728 (Experimental), February 2007.

[13] E.P.C. Jones, L. Li, J.K. Schmidtke, and P.A.S. Ward. Practical routing in delay-tolerant networks. *IEEE Transactions on Mobile Computing*, 6(8):943 –959, Aug 2007.

[14] Rudolph Emil Kalman. A new approach to linear filtering and prediction problems. *Transactions of the ASME–Journal of Basic Engineering*, 82(Series D):35–45, 1960.

[15] J. Leguay, T. Friedman, and V. Conan. Evaluating mobility pattern space routing for dtns. In *INFOCOM 2006. 25th IEEE International Conference on Computer Communications. Proceedings*, pages 1 –10, April 2006.

[16] Morten Lindeberg, Stein Kristiansen, Vera Goebel, and Thomas Plagemann. Mac layer support for delay tolerant video transport in disruptive manets. In *Proceedings of the 10th international IFIP TC 6 conference on Networking - Volume Part I*, NETWORKING'11, pages 106–119, Berlin, Heidelberg, 2011. Springer-Verlag.

[17] Morten Lindeberg, Stein Kristiansen, Thomas Plagemann, and Vera Goebel. Challenges and techniques for video streaming over mobile ad hoc networks. *Springer Multimedia Systems*, 17:1–32, 2010.

[18] Anders Lindgren, Avri Doria, and Olov Scheln. Probabilistic routing in intermittently connected networks. In Petre Dini, Pascal Lorenz, and Jos Neuman de Souza, editors, *Service Assurance with Partial and Intermittent Resources*, volume 3126 of *Lecture Notes in Computer Science*, pages 239–254. Springer Berlin / Heidelberg, 2004.

[19] Cong Liu and Jie Wu. An optimal probabilistic forwarding protocolin delay tolerant networks. In *Proceedings of the tenth ACM international symposium on Mobile ad hoc networking and computing*, MobiHoc '09, pages 105–114, New York, NY, USA, 2009. ACM.

[20] M. Musolesi, S. Hailes, and C. Mascolo. Adaptive routing for intermittently connected mobile ad hoc networks. In *Sixth IEEE International Symposium on a World of Wireless Mobile and Multimedia Networks, 2005. WoWMoM 2005.*, pages 183 – 189, June 2005.

[21] Hideya Ochiai, Kenichi Shimotada, and Hiroshi Esaki. Dtipn: delay tolerant ip networking for opportunistic network applications. In *Proceedings of the Second International Workshop on Mobile Opportunistic Networking*, MobiOpp '10, pages 65–71, New York, NY, USA, 2010. ACM.

[22] David Palma and Marilia Curado. Inside-out olsr scalability analysis. In *Proceedings of the 8th International Conference on Ad-Hoc, Mobile and Wireless Networks*, ADHOC-NOW '09, pages 354–359, Berlin, Heidelberg, 2009. Springer-Verlag.

[23] C. Perkins, E. Belding-Royer, and S. Das. RFC 3561: Ad hoc On-Demand Distance Vector (AODV) Routing. RFC 3561 (Experimental), July 2003.

[24] Thomas Plagemann, Vera Goebel, Ellen Munthe-Kaas, Knut Omang, Xabiel Garca Paeda, and Matti Siekkinen. Dtstream: Delay tolerant streaming services, January 2008. http://www.mn.uio.no/ifi/english/research/projects/dtstream/.

[25] Hany Samuel, Weihua Zhuang, and Bruno Preiss. Dtn based dominating set routing technique for mobile ad hoc networks. In *Proceedings of the 5th International ICST Conference on Heterogeneous Networking for Quality, Reliability, Security and Robustness*, QShine '08, pages 9:1–9:7, ICST, Brussels, Belgium, Belgium, 2008. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering).

[26] Norun Sanderson, Katrine Stemland Skjelsvik, Ovidiu Valentin Drugan, Matija Puzar, Vera Goebel, Ellen Munthe-Kaas, and Thomas Plagemann.

Developing mobile middleware - an analysis of rescue and emergency operations. Technical Report Research Report no.358, Department of informatics, University of Oslo, Norway, June 2007.

[27] Thrasyvoulos Spyropoulos, Konstantinos Psounis, and Cauligi S. Raghavendra. Spray and wait: an efficient routing scheme for intermittently connected mobile networks. In *Proceedings of the 2005 ACM SIGCOMM workshop on Delay-tolerant networking*, WDTN '05, pages 252–259, New York, NY, USA, 2005. ACM.

[28] Kun Tan, Qian Zhang, and Wenwu Zhu. Shortest path routing in partially connected ad hoc networks. In *Global Telecommunications Conference, 2003. GLOBECOM '03. IEEE*, volume 2, pages 1038 – 1042 Vol.2, dec. 2003.

[29] Amin Vahdat and David Becker. Epidemic routing for partially-connected ad hoc networks. *Duke Technical Report CS-2000-06*, 2000.

[30] Various. igraph. http://igraph.sourceforge.net.

[31] Various. Ns3 network simulator. http://www.nsnam.org/.

[32] Ming-Jun Xiao, Liu-Sheng Huang, Qun-Feng Dong, An Liu, and Zhen-Guo Yang. Leapfrog: Optimal opportunistic routing in probabilistically contacted delay tolerant networks. *Journal of Computer Science and Technology*, 24:975–986, 2009.

[33] W. Zhao, M. Ammar, and E. Zegura. A message ferrying approach for data delivery in sparse mobile ad hoc networks. In *Proceedings of the 5th ACM international symposium on Mobile ad hoc networking and computing*, MobiHoc '04, pages 187–198, New York, NY, USA, 2004.

Appendix A

Evaluation Graphs

In Chapter 7, we evaluated DSMC. For the performance study, we ran 18 configurations for Scenario 1 in 5 runs. Additionally, we ran 2 configurations for Scenario 2 in 5 runs. The results of these were presented in the evaluation, however not all of them in detail. In this appendix, we show all the graphs from all the runs. This includes graphs for packet delivery (PD) and cumulative distribution functions (CDF) for delay.

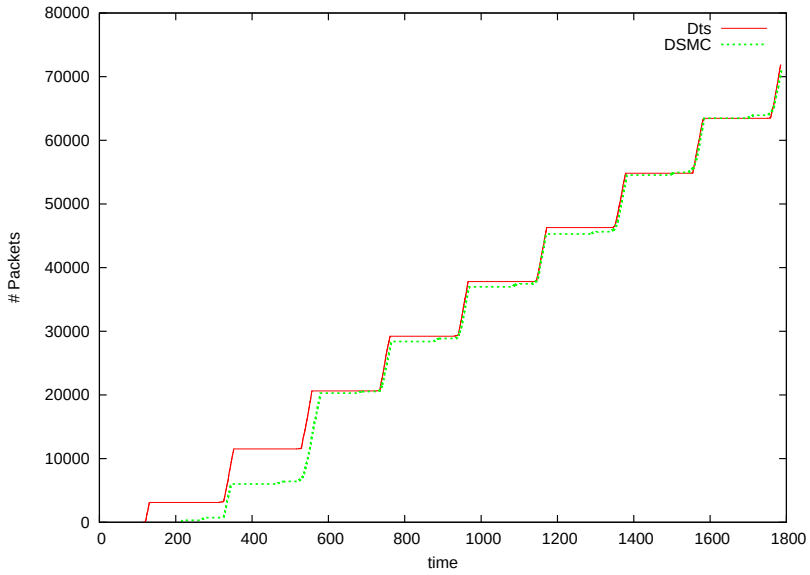

Figure A.1: Single Run Scenario 1 Configuration 1, Seed 1

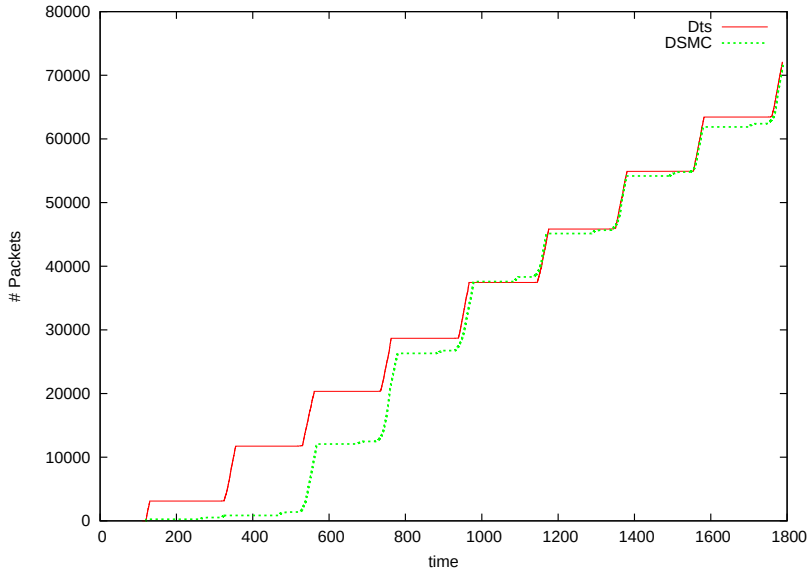

Figure A.2: Single Run Scenario 1 Configuration 1, Seed 2

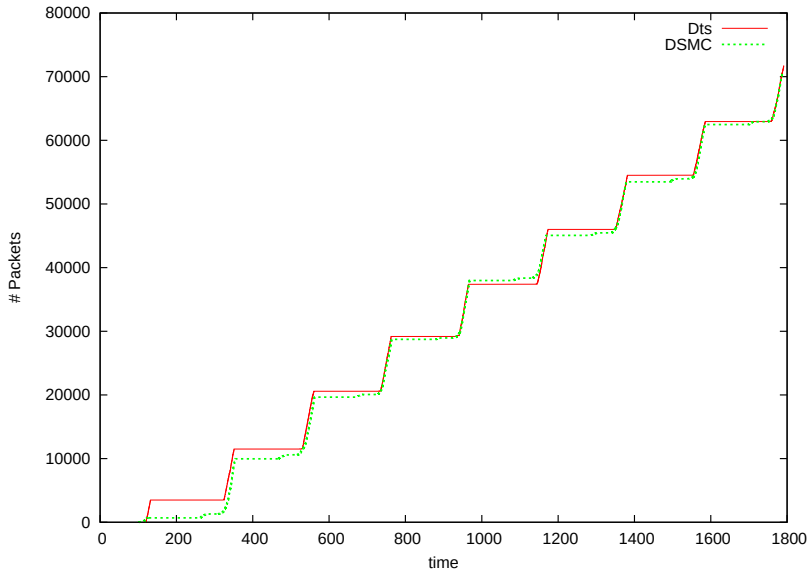

Figure A.3: Single Run Scenario 1 Configuration 1, Seed 3

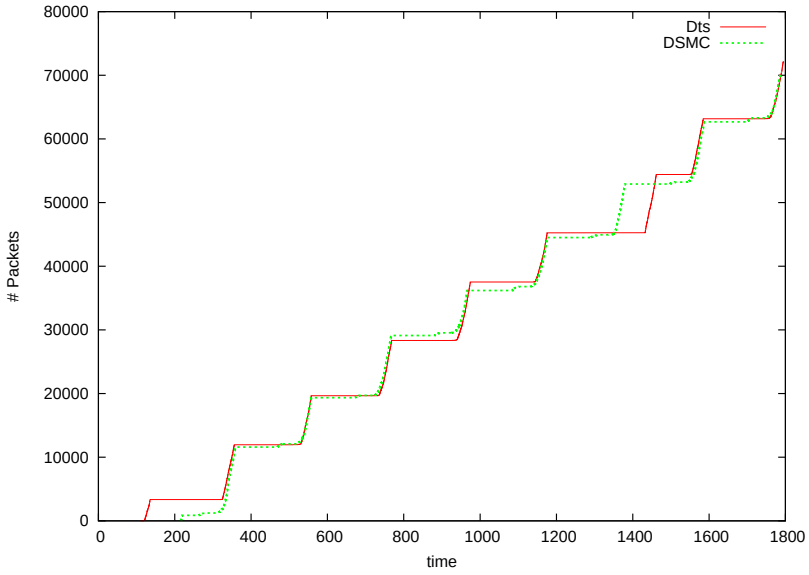

Figure A.4: Single Run Scenario 1 Configuration 1, Seed 4

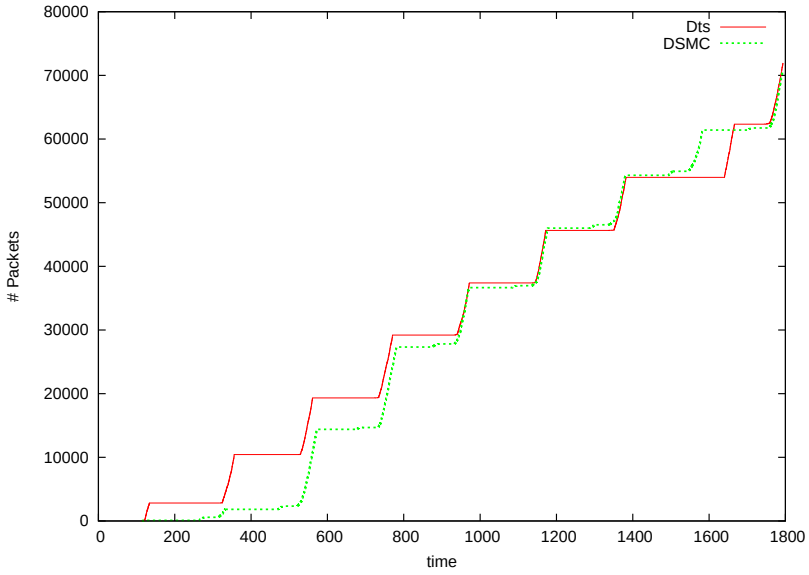

Figure A.5: Single Run Scenario 1 Configuration 1, Seed 5

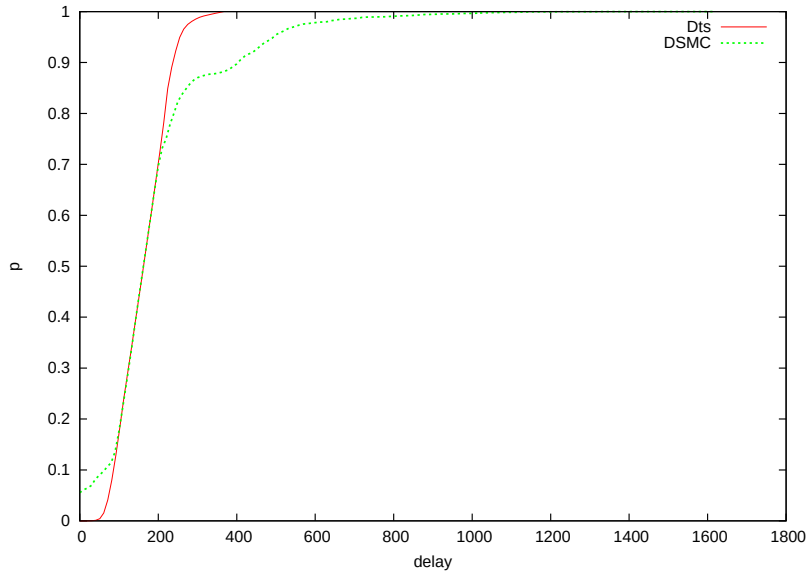

Figure A.6: CDF for All Runs With Configuration 1 in Scenario 1

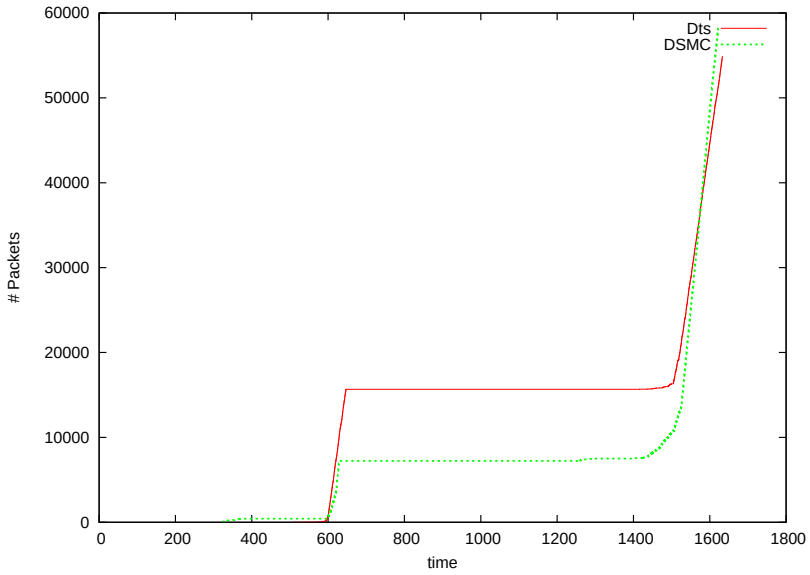

Figure A.7: Single Run Scenario 1 Configuration 2, Seed 1

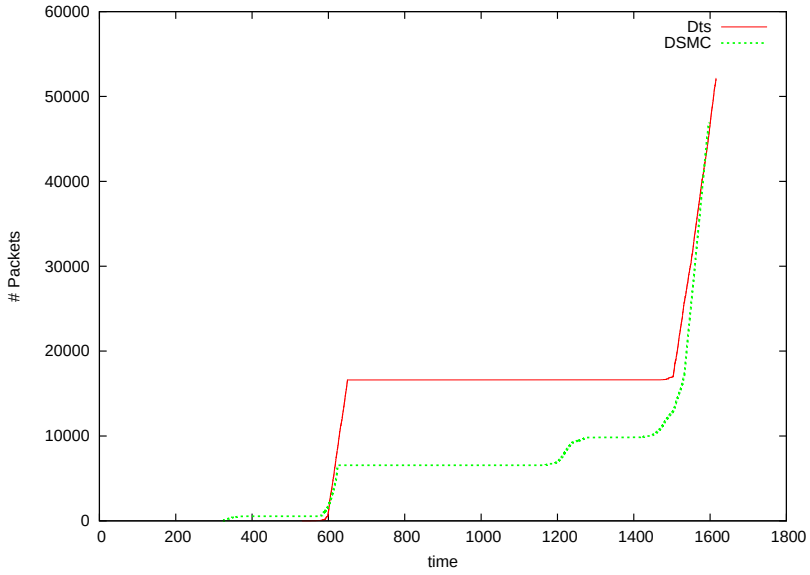

Figure A.8: Single Run Scenario 1 Configuration 2, Seed 2

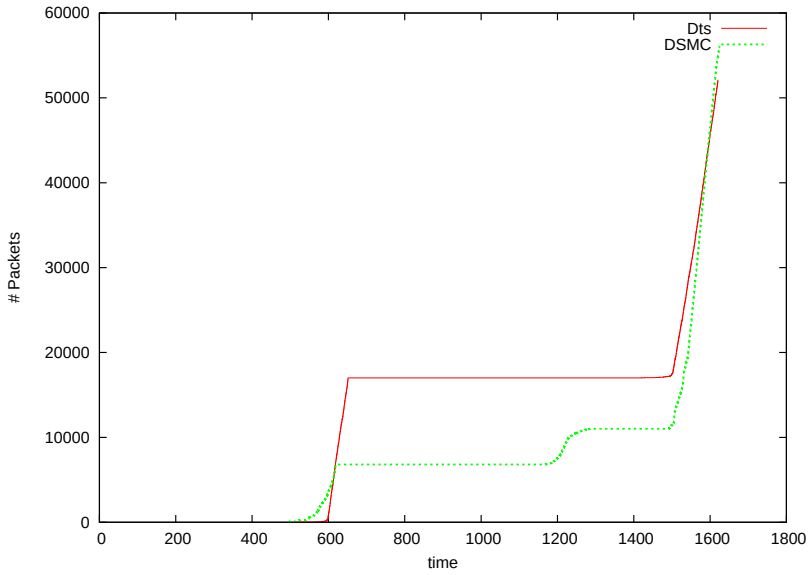

Figure A.9: Single Run Scenario 1 Configuration 2, Seed 3

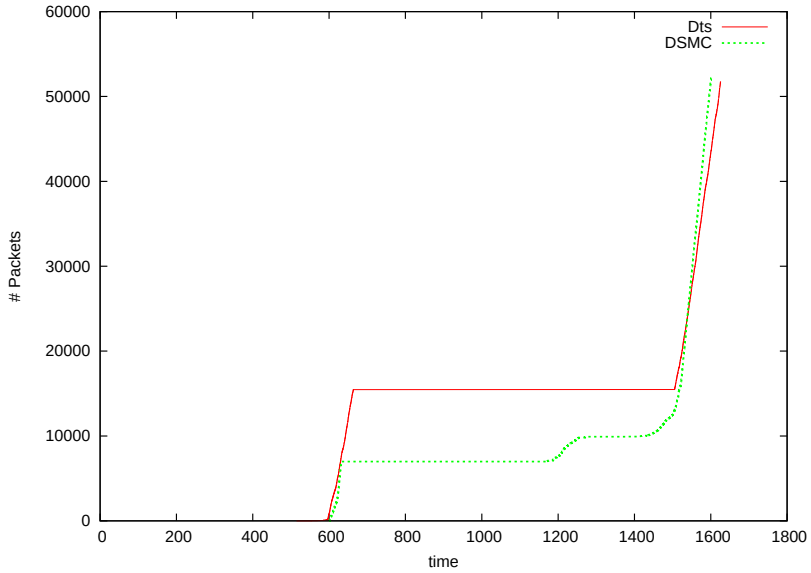

Figure A.10: Single Run Scenario 1 Configuration 2, Seed 4

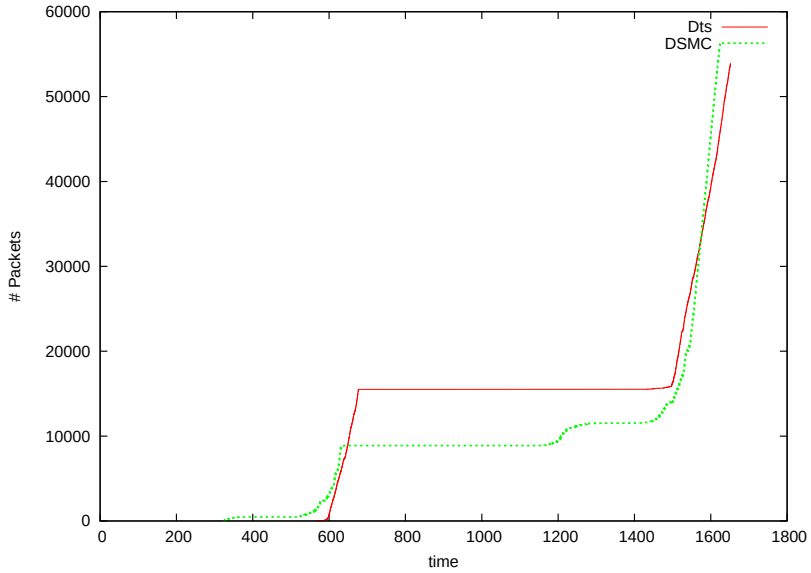

Figure A.11: Single Run Scenario 1 Configuration 2, Seed 5

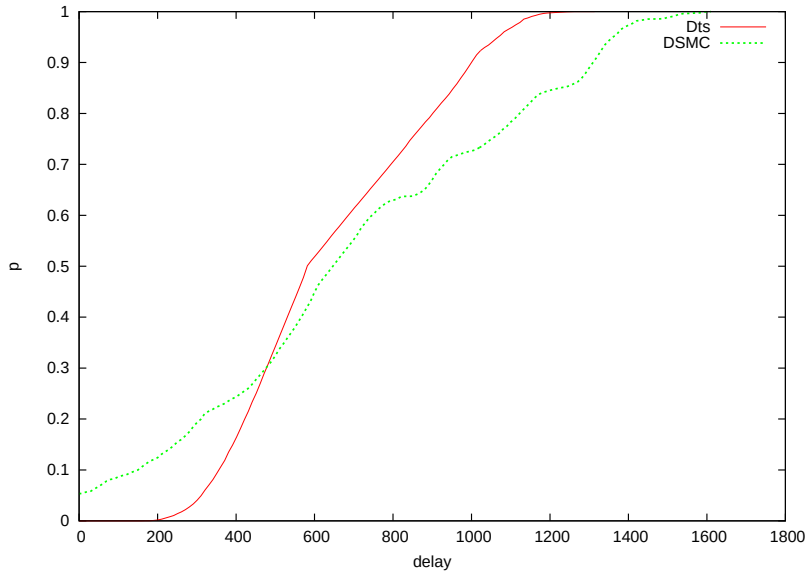

Figure A.12: CDF for All Runs With Configuration 2 in Scenario 1

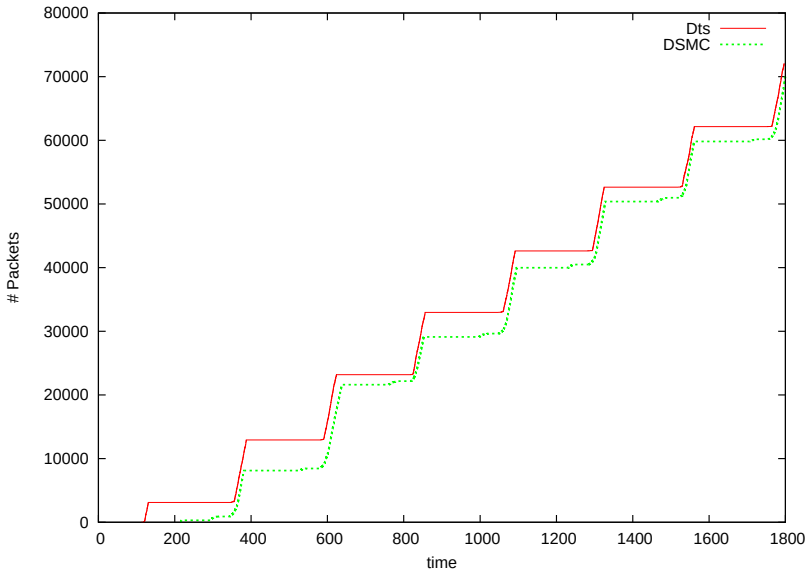

Figure A.13: Single Run Scenario 1 Configuration 3, Seed 1

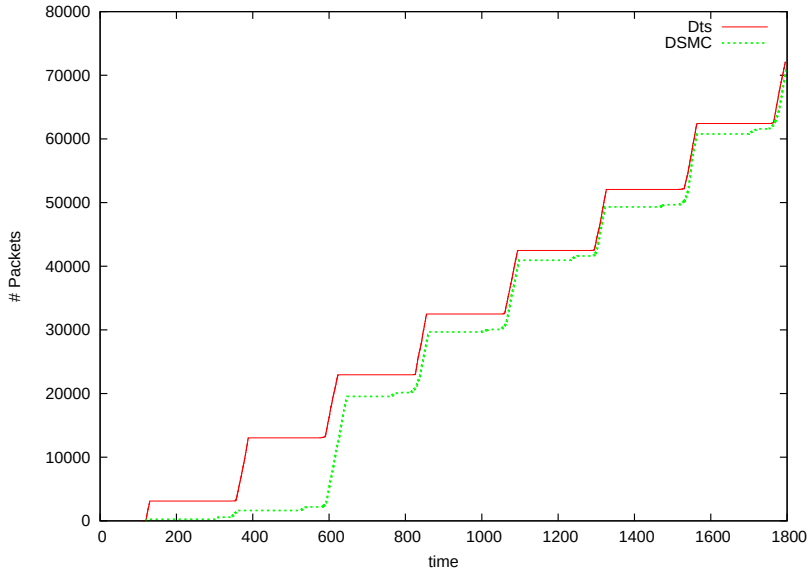

Figure A.14: Single Run Scenario 1 Configuration 3, Seed 2

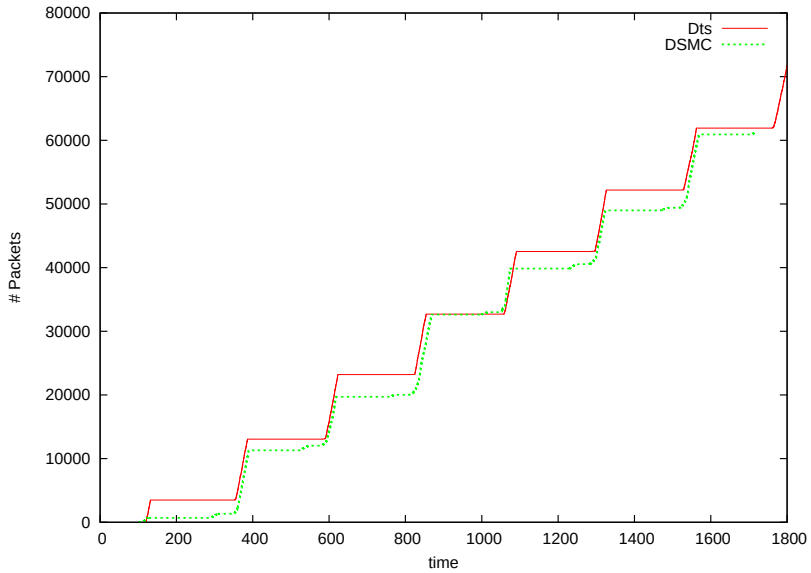

Figure A.15: Single Run Scenario 1 Configuration 3, Seed 3

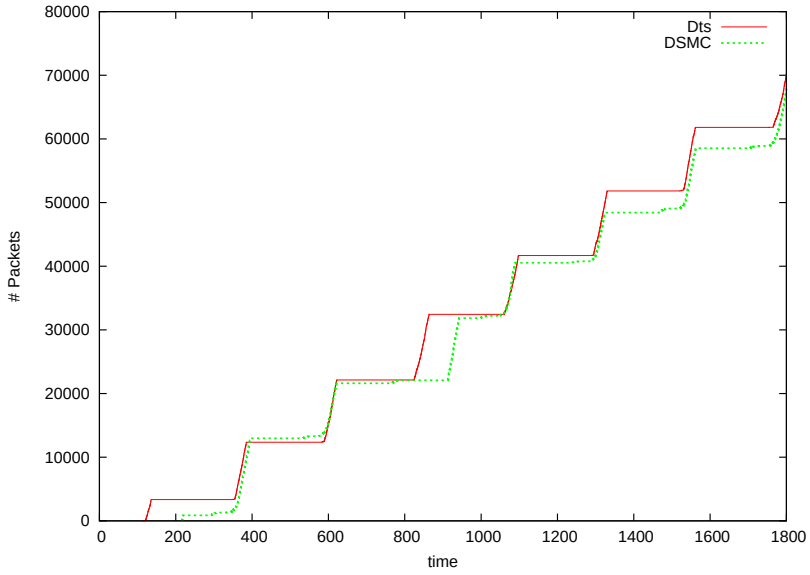

Figure A.16: Single Run Scenario 1 Configuration 3, Seed 4

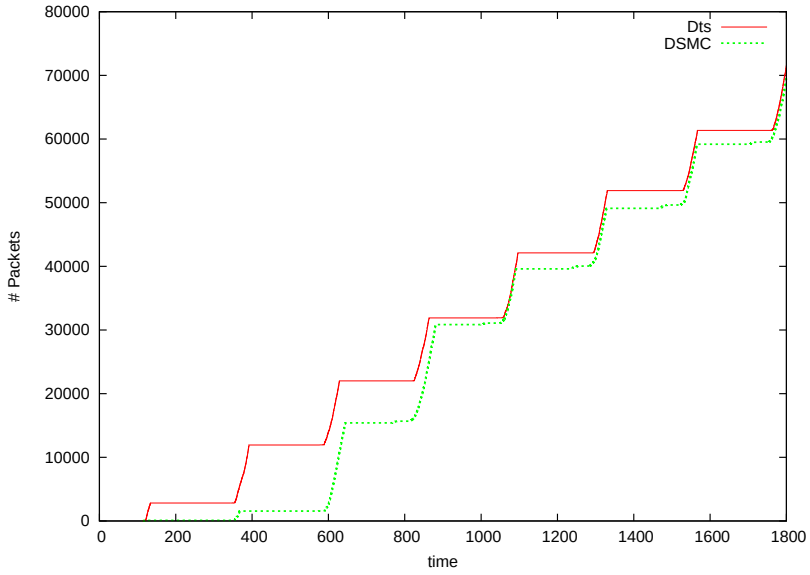

Figure A.17: Single Run Scenario 1 Configuration 3, Seed 5

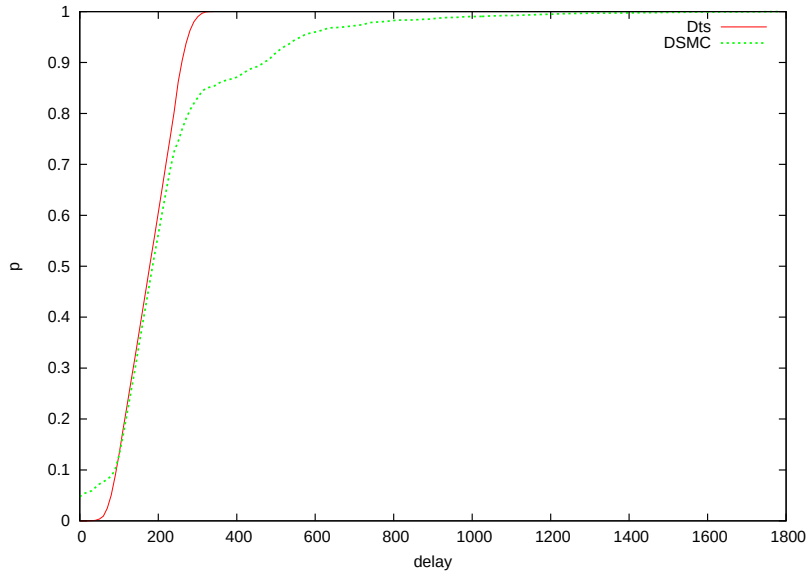

Figure A.18: CDF for All Runs With Configuration 3 in Scenario 1

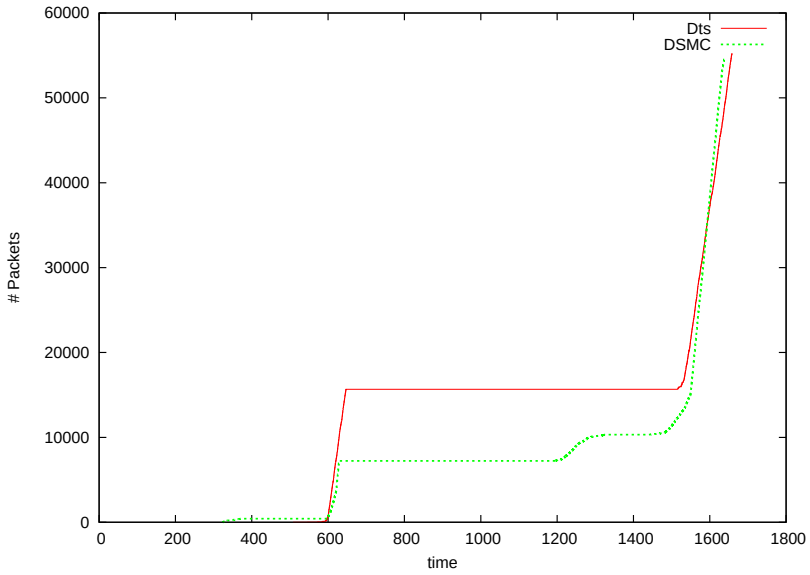

Figure A.19: Single Run Scenario 1 Configuration 4, Seed 1

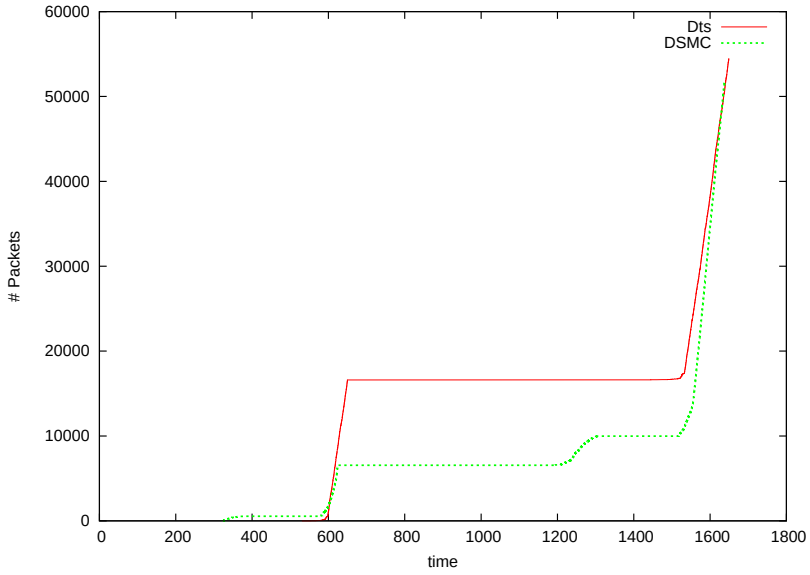

Figure A.20: Single Run Scenario 1 Configuration 4, Seed 2

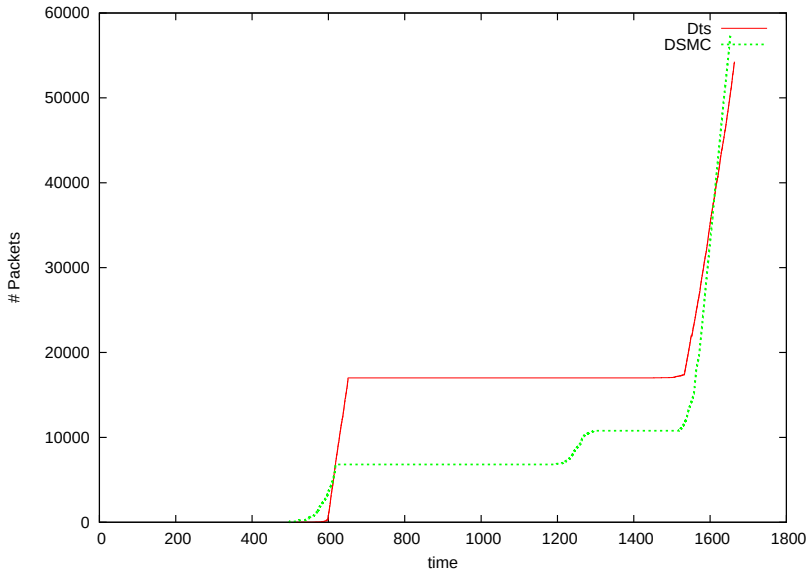

Figure A.21: Single Run Scenario 1 Configuration 4, Seed 3

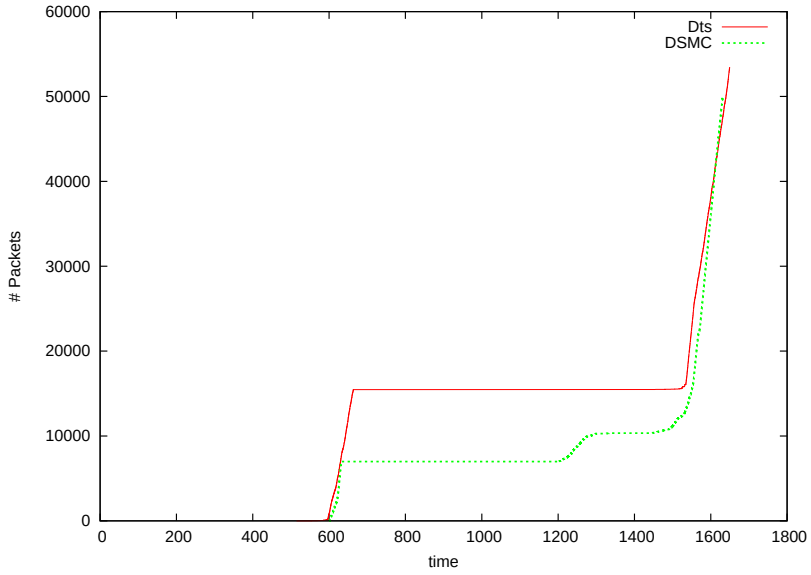

Figure A.22: Single Run Scenario 1 Configuration 4, Seed 4

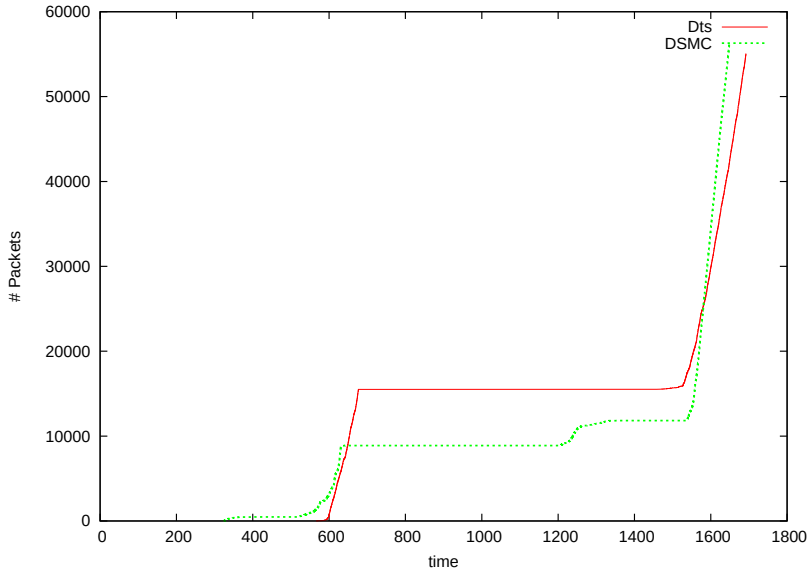

Figure A.23: Single Run Scenario 1 Configuration 4, Seed 5

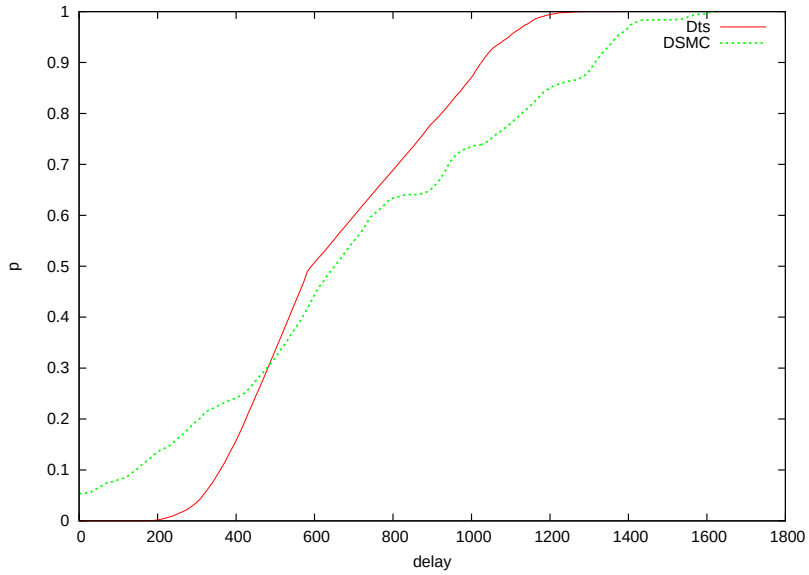

Figure A.24: CDF for All Runs With Configuration 4 in Scenario 1

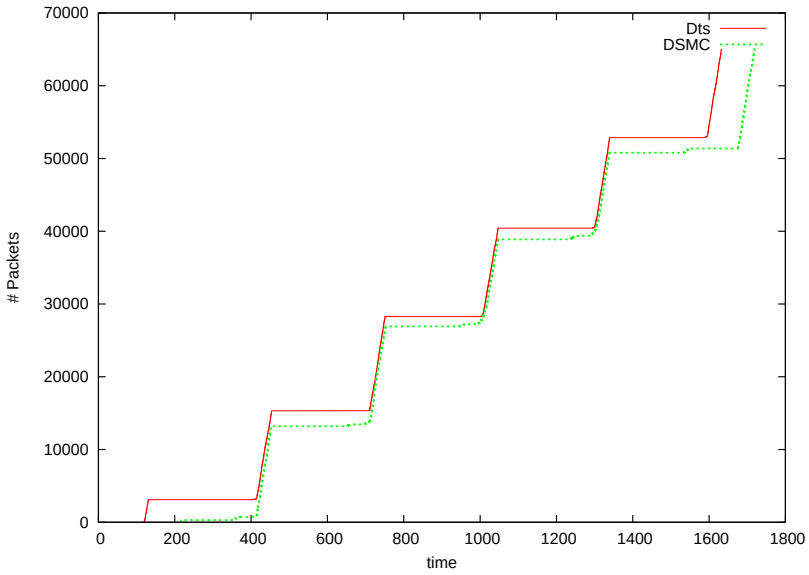

Figure A.25: Single Run Scenario 1 Configuration 5, Seed 1

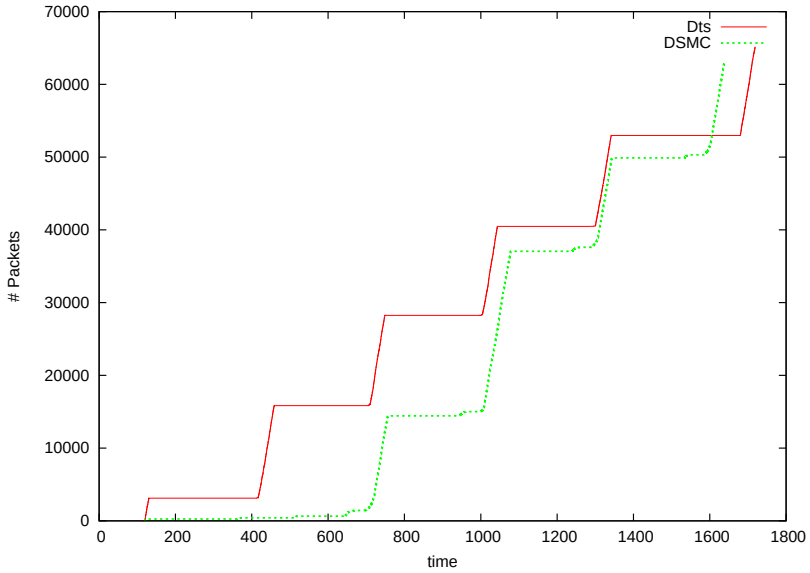

Figure A.26: Single Run Scenario 1 Configuration 5, Seed 2

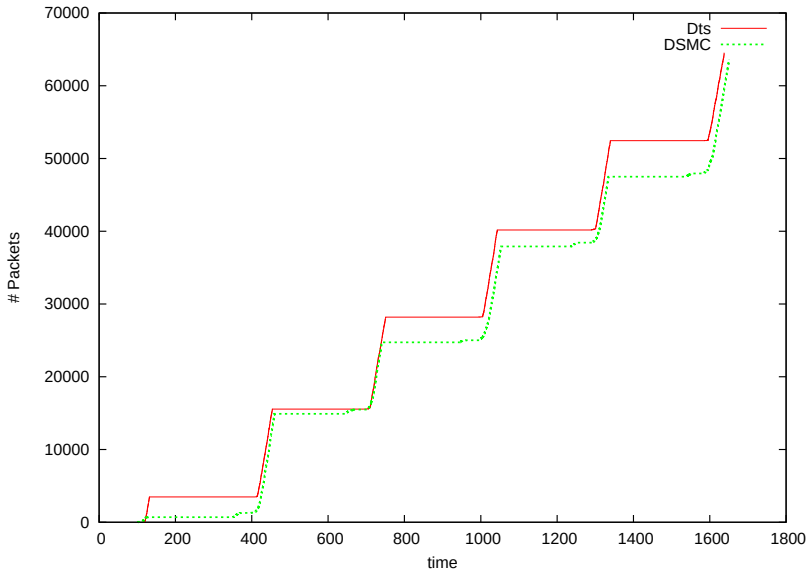

Figure A.27: Single Run Scenario 1 Configuration 5, Seed 3

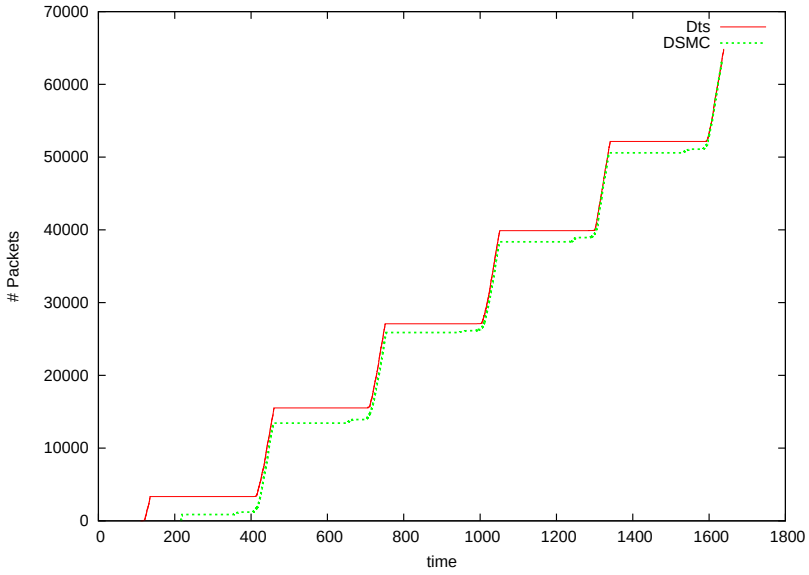

Figure A.28: Single Run Scenario 1 Configuration 5, Seed 4

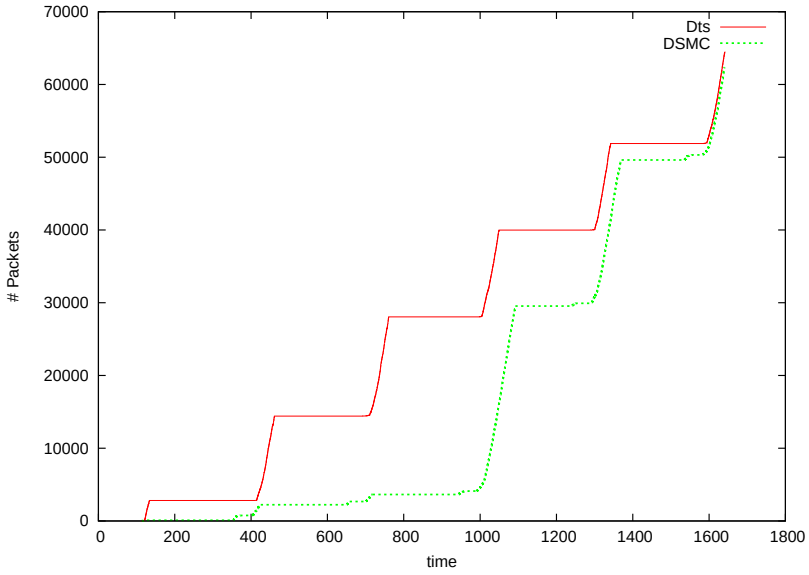

Figure A.29: Single Run Scenario 1 Configuration 5, Seed 5

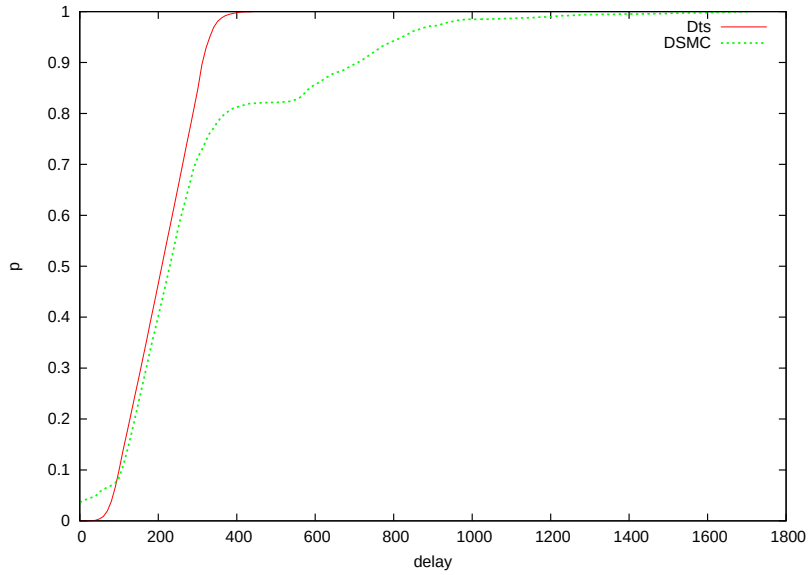

Figure A.30: CDF for All Runs With Configuration 5 in Scenario 1

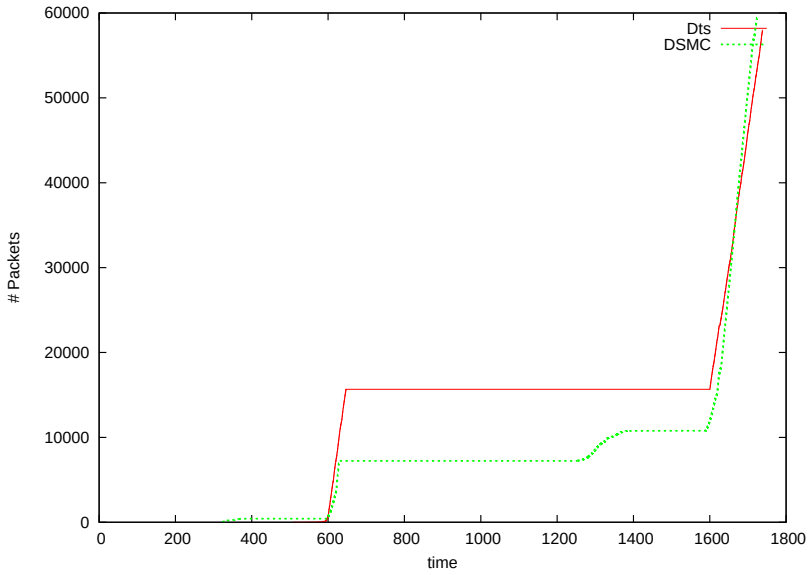

Figure A.31: Single Run Scenario 1 Configuration 6, Seed 1

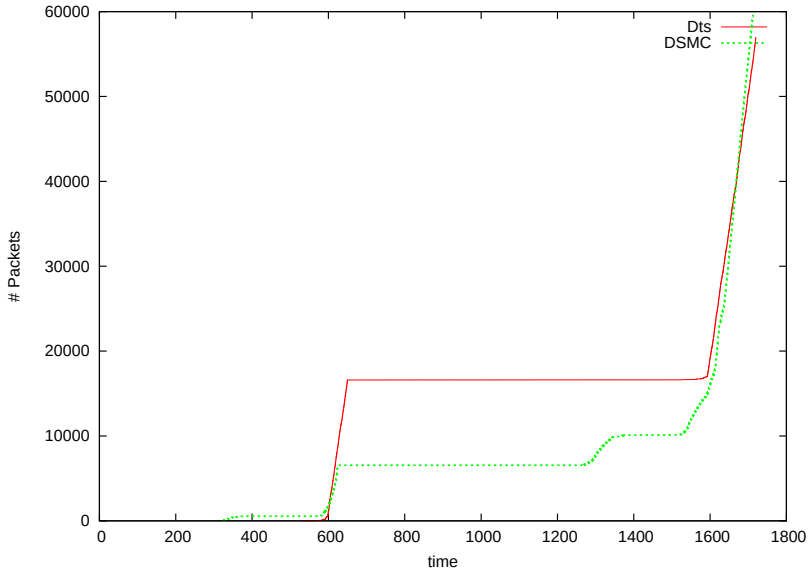

Figure A.32: Single Run Scenario 1 Configuration 6, Seed 2

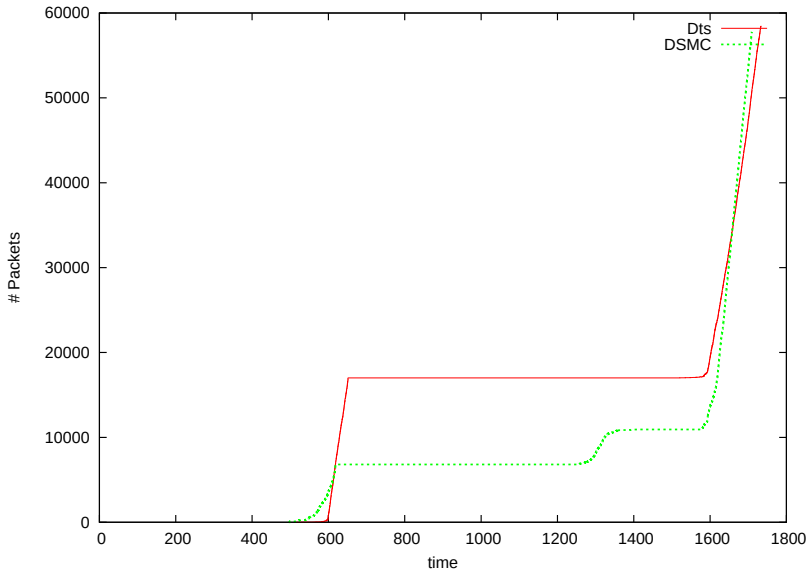

Figure A.33: Single Run Scenario 1 Configuration 6, Seed 3

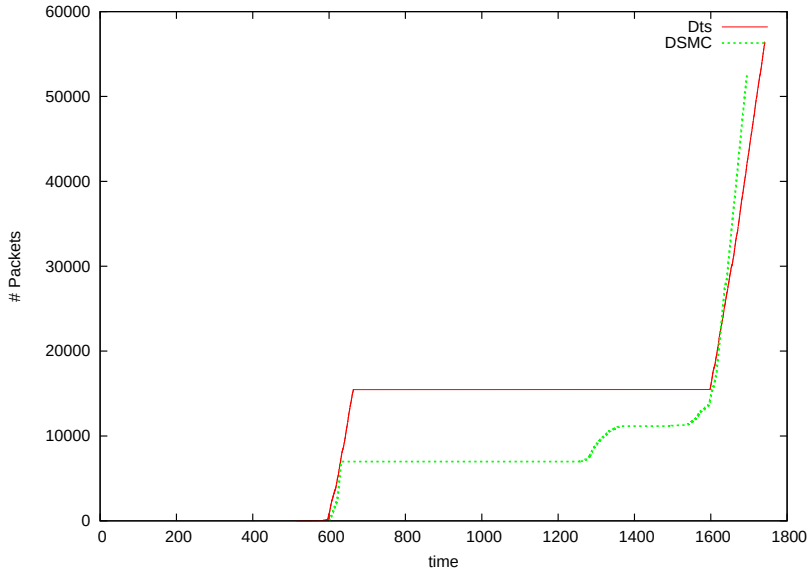

Figure A.34: Single Run Scenario 1 Configuration 6, Seed 4

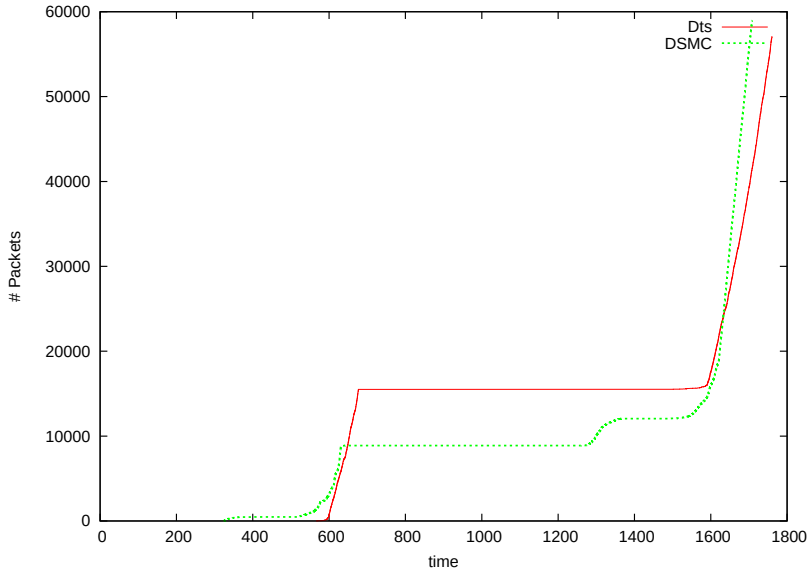

Figure A.35: Single Run Scenario 1 Configuration 6, Seed 5

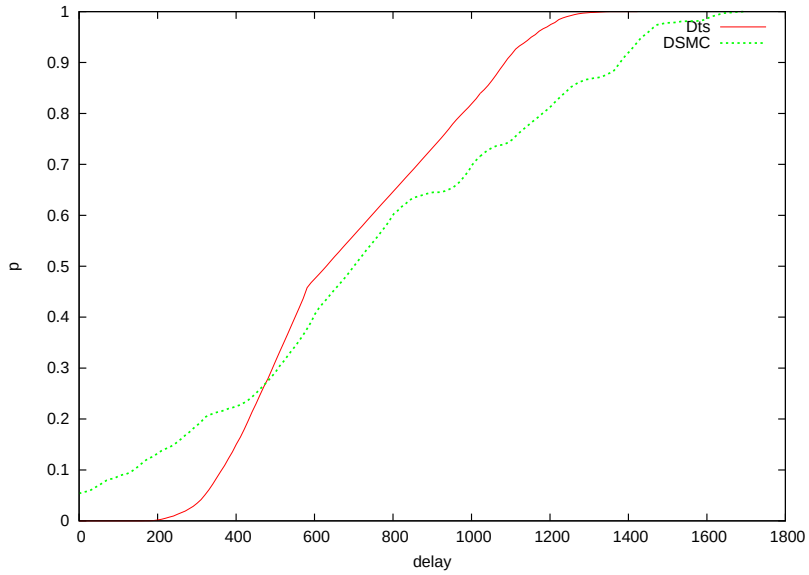

Figure A.36: CDF for All Runs With Configuration 6 in Scenario 1

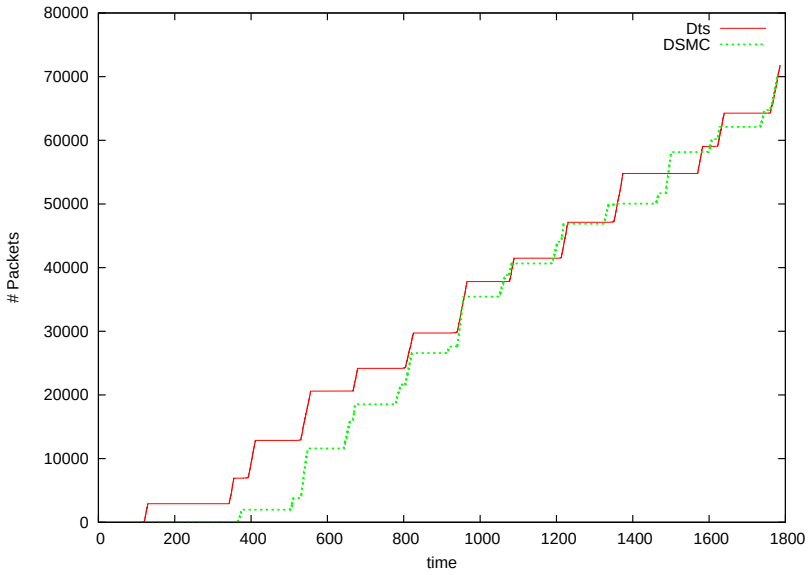

Figure A.37: Single Run Scenario 1 Configuration 7, Seed 1

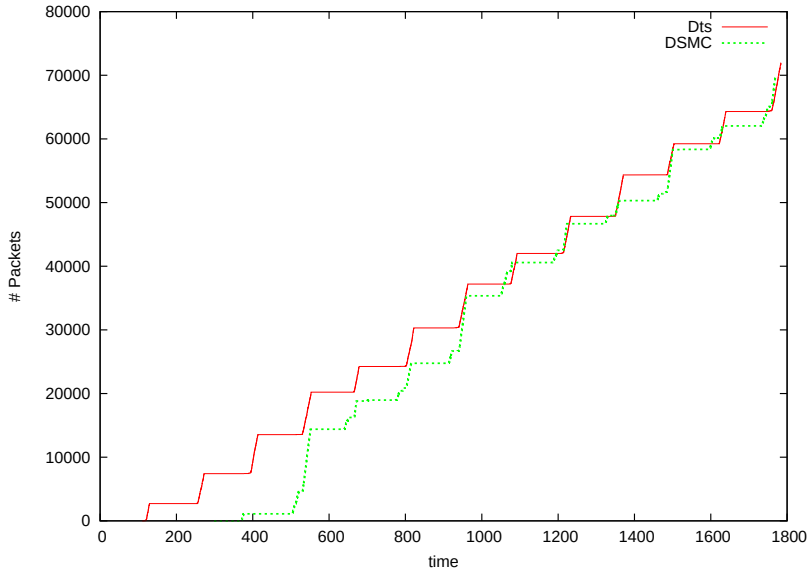

Figure A.38: Single Run Scenario 1 Configuration 7, Seed 2

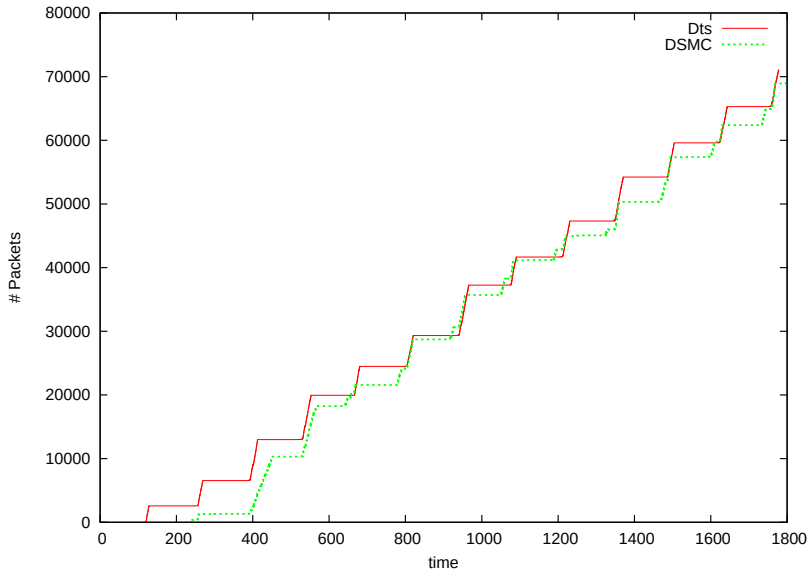

Figure A.39: Single Run Scenario 1 Configuration 7, Seed 3

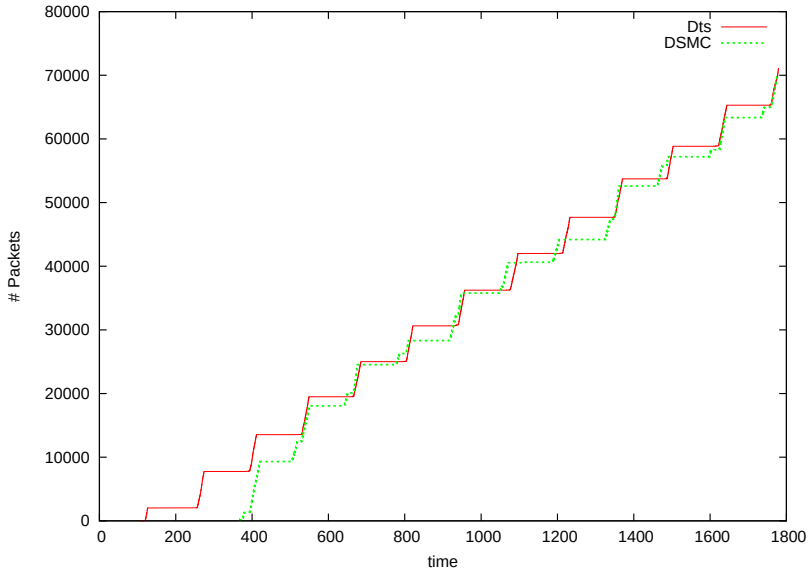

Figure A.40: Single Run Scenario 1 Configuration 7, Seed 4

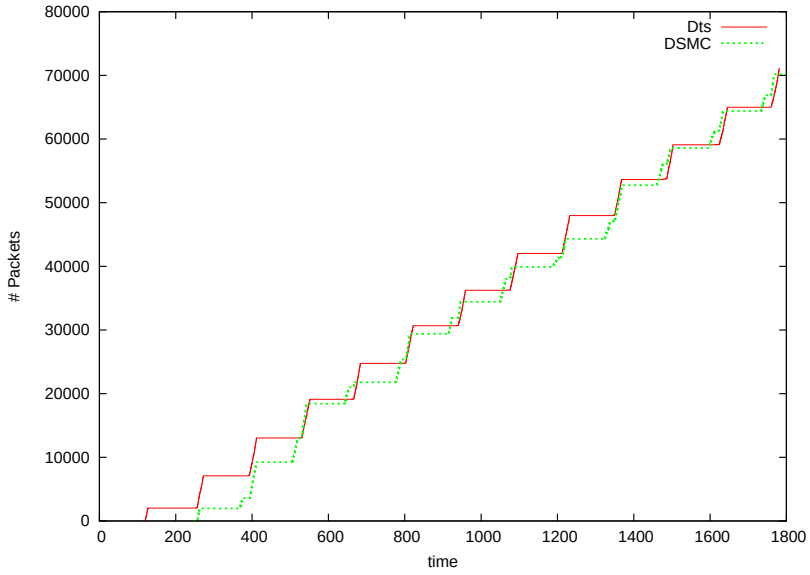

Figure A.41: Single Run Scenario 1 Configuration 7, Seed 5

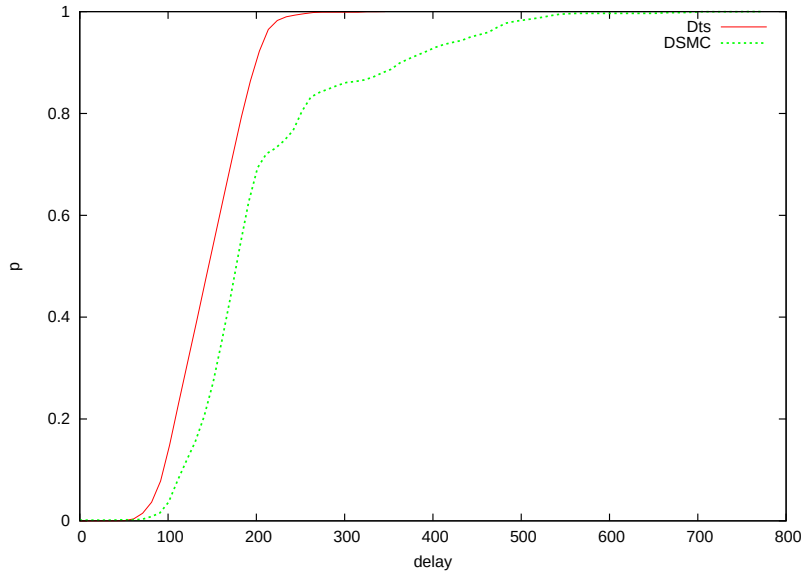

Figure A.42: CDF for All Runs With Configuration 7 in Scenario 1

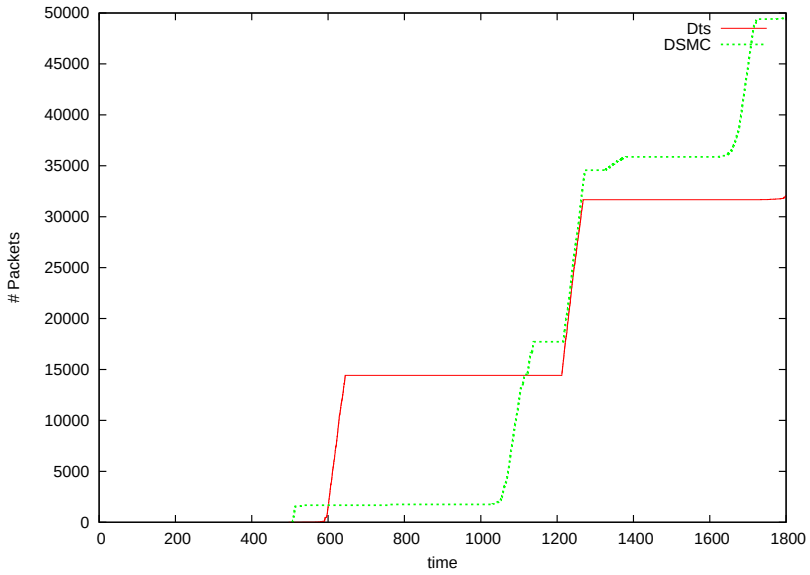

Figure A.43: Single Run Scenario 1 Configuration 8, Seed 1

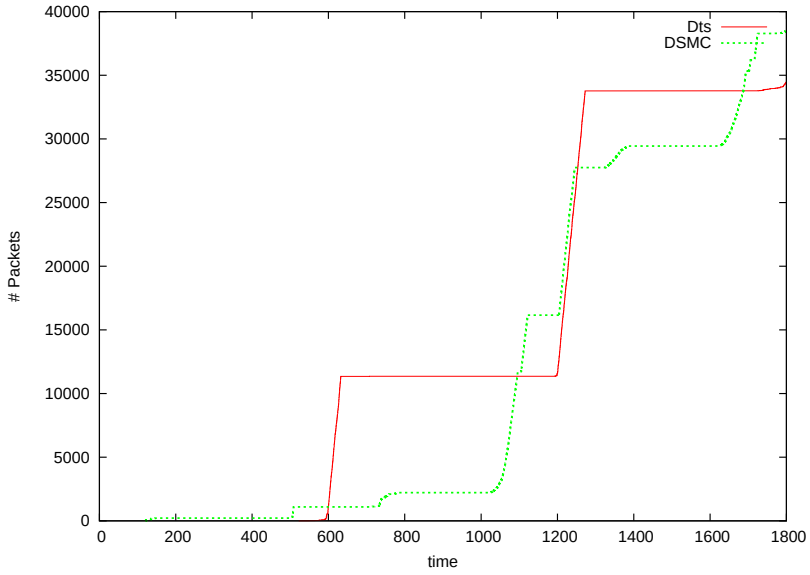

Figure A.44: Single Run Scenario 1 Configuration 8, Seed 2

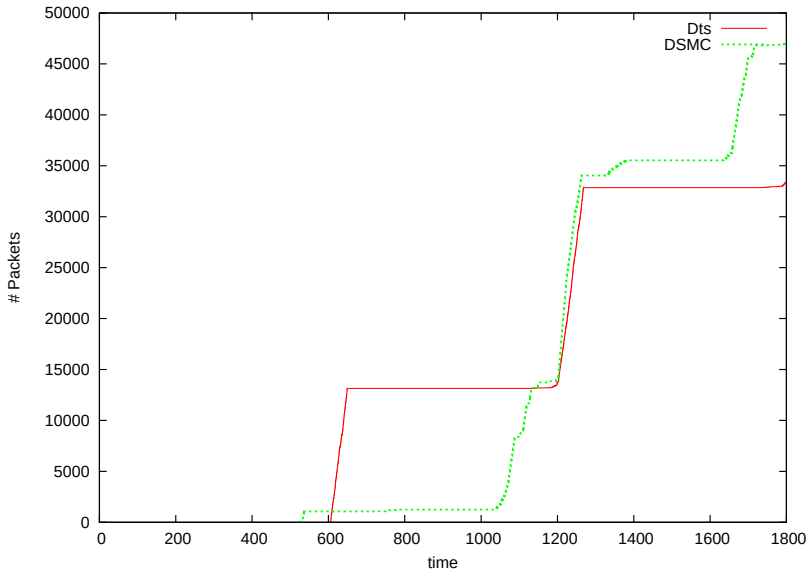

Figure A.45: Single Run Scenario 1 Configuration 8, Seed 3

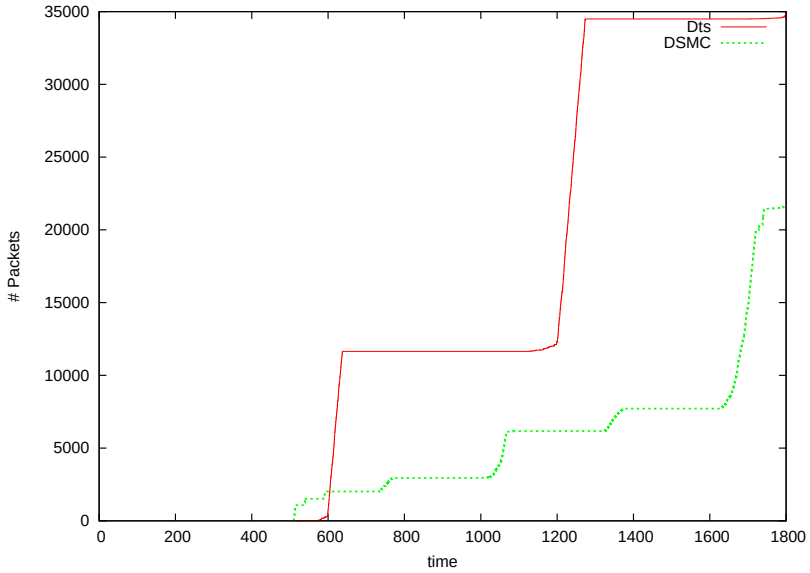

Figure A.46: Single Run Scenario 1 Configuration 8, Seed 4

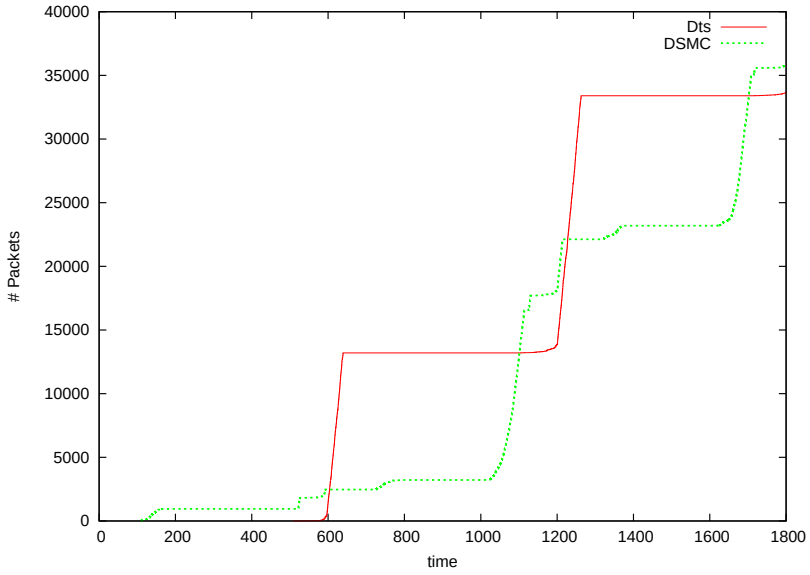

Figure A.47: Single Run Scenario 1 Configuration 8, Seed 5

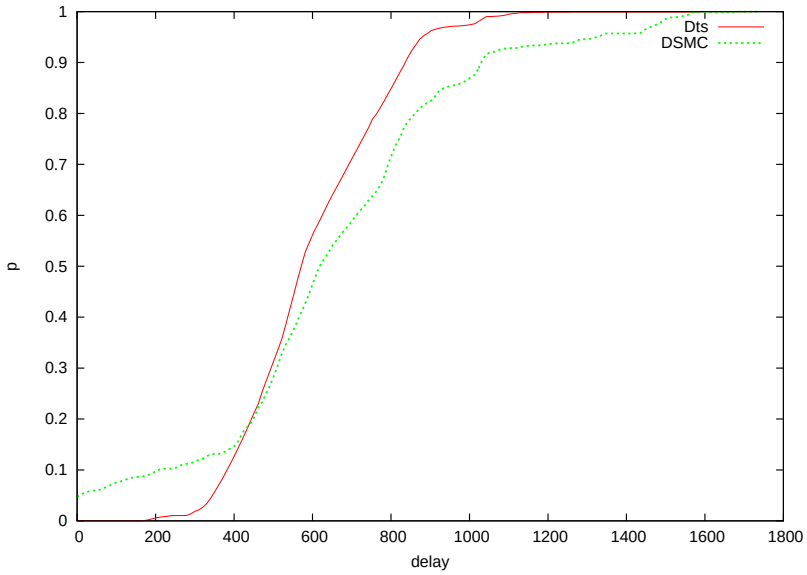

Figure A.48: CDF for All Runs With Configuration 8 in Scenario 1

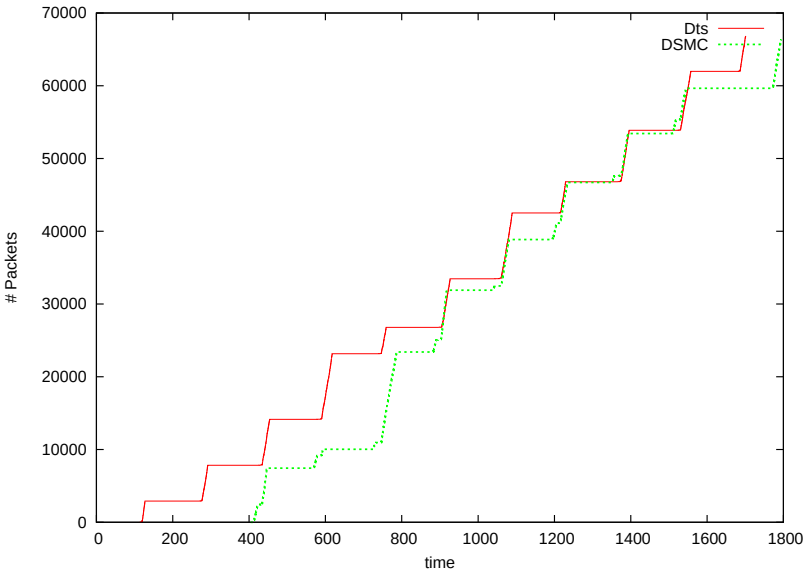

Figure A.49: Single Run Scenario 1 Configuration 9, Seed 1

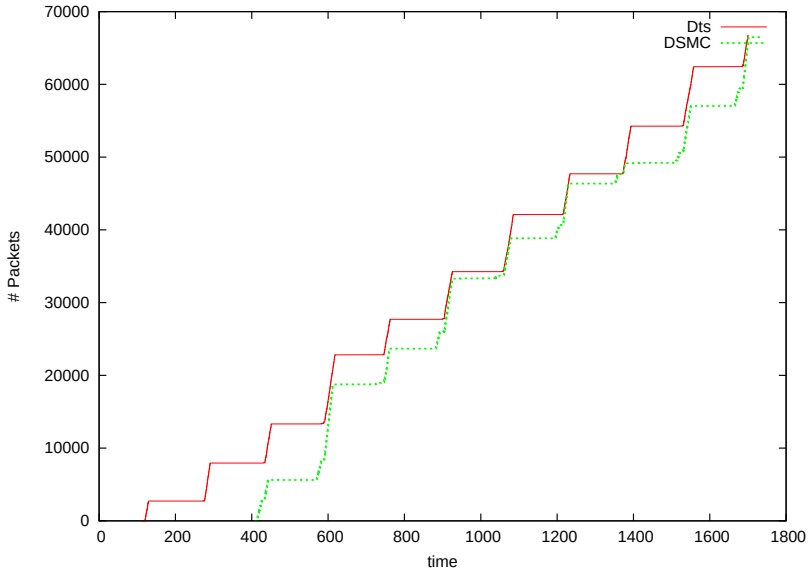

Figure A.50: Single Run Scenario 1 Configuration 9, Seed 2

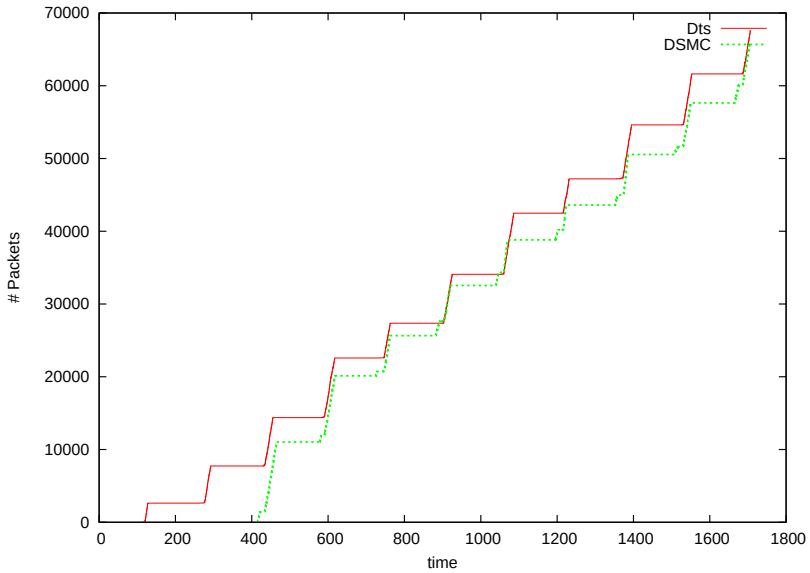

Figure A.51: Single Run Scenario 1 Configuration 9, Seed 3

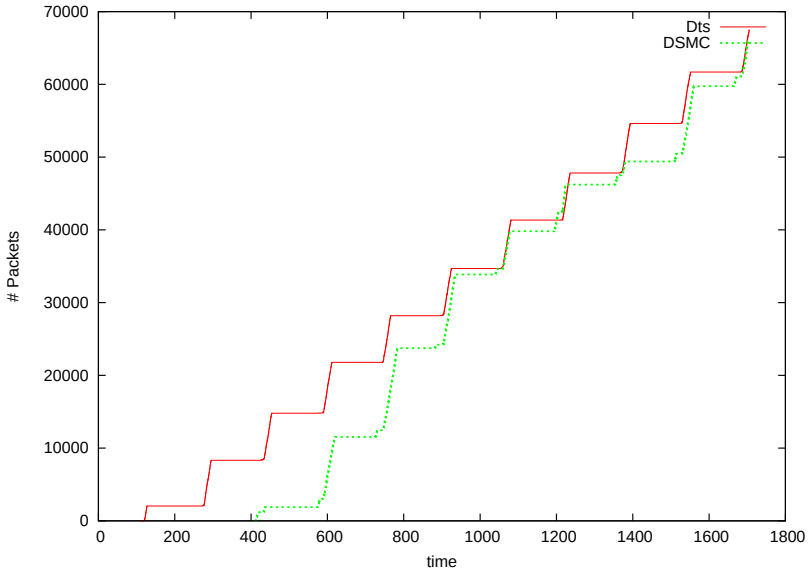

Figure A.52: Single Run Scenario 1 Configuration 9, Seed 4

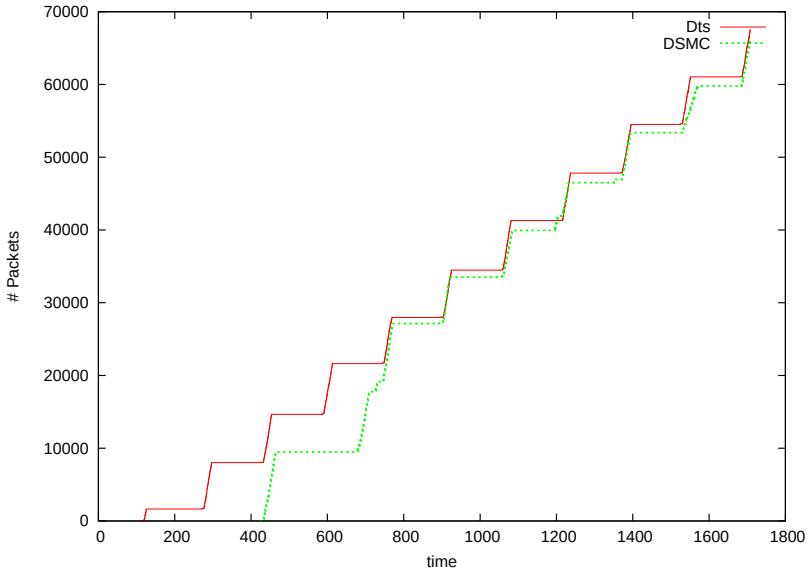

Figure A.53: Single Run Scenario 1 Configuration 9, Seed 5

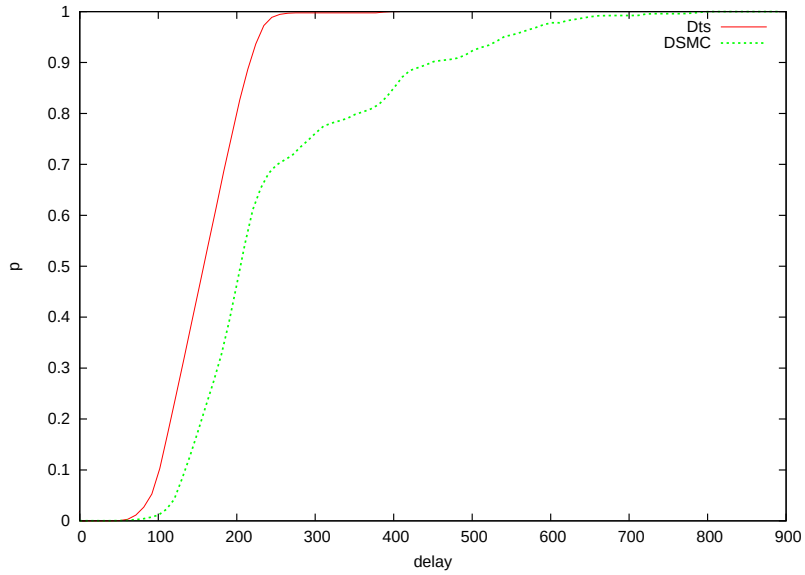

Figure A.54: CDF for All Runs With Configuration 9 in Scenario 1

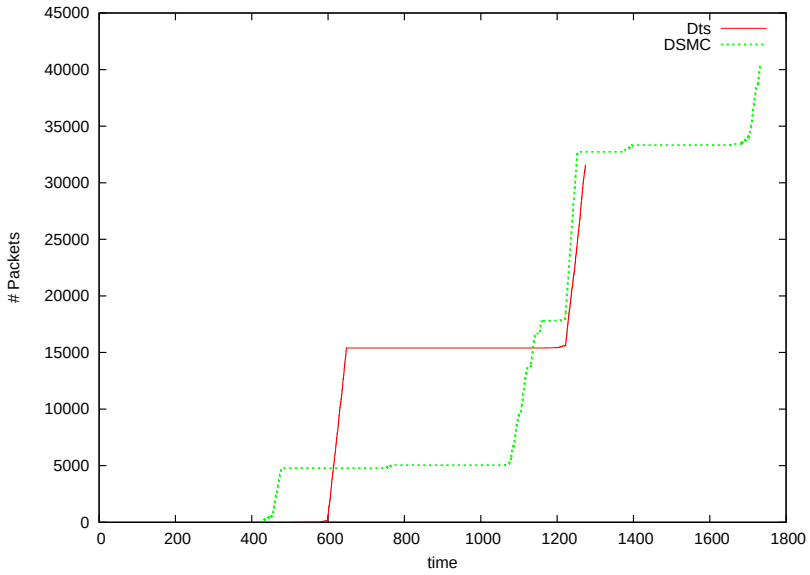

Figure A.55: Single Run Scenario 1 Configuration 10, Seed 1

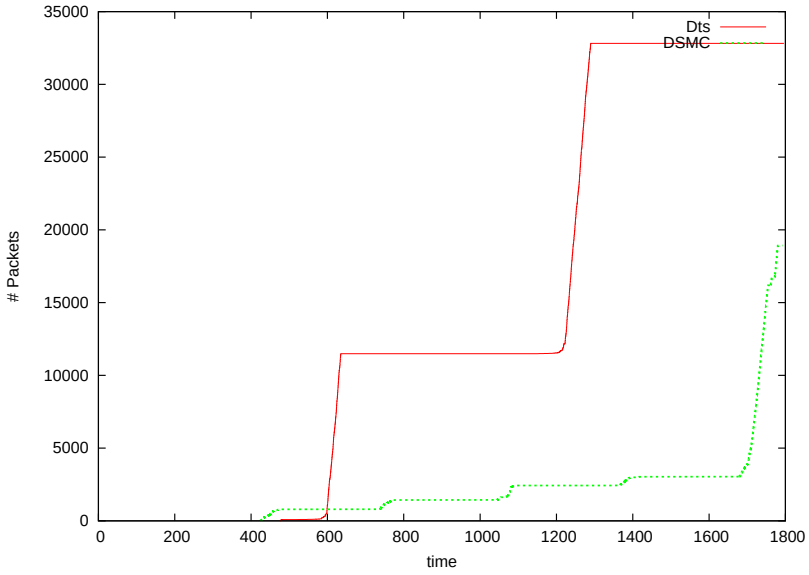

Figure A.56: Single Run Scenario 1 Configuration 10, Seed 2

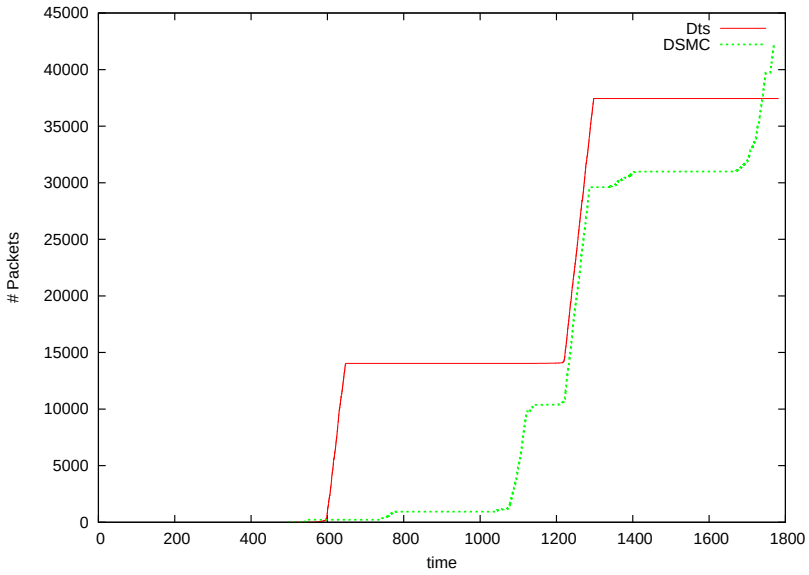

Figure A.57: Single Run Scenario 1 Configuration 10, Seed 3

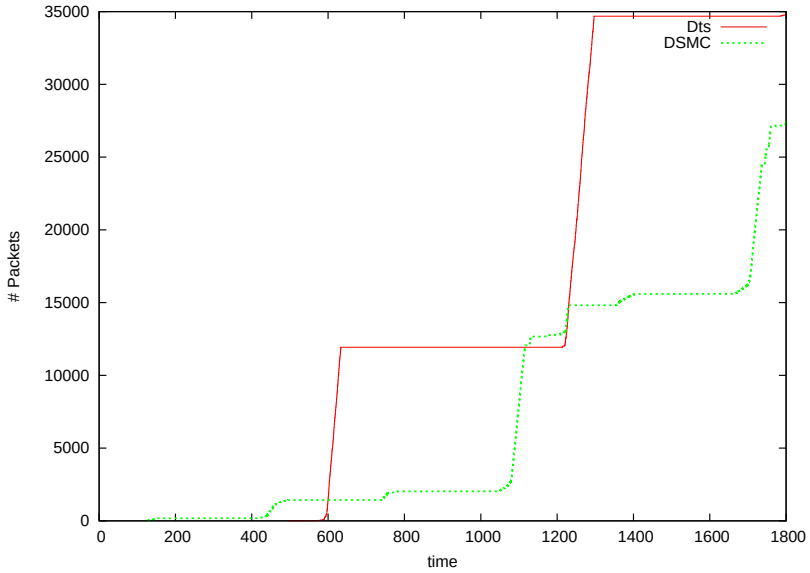

Figure A.58: Single Run Scenario 1 Configuration 10, Seed 4

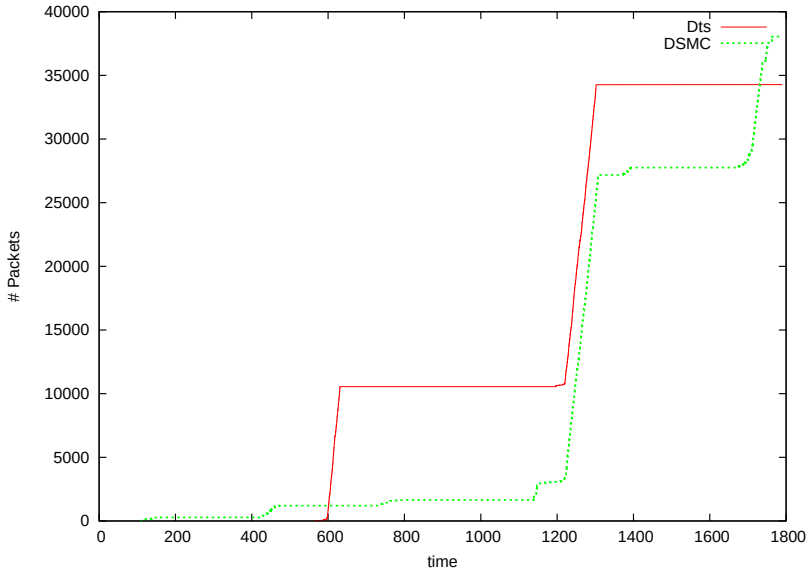

Figure A.59: Single Run Scenario 1 Configuration 10, Seed 5

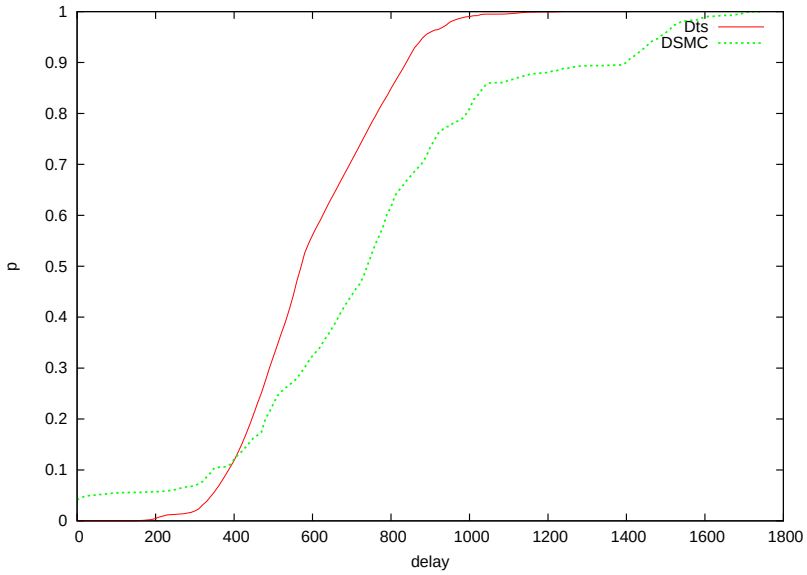

Figure A.60: CDF for All Runs With Configuration 10 in Scenario 1

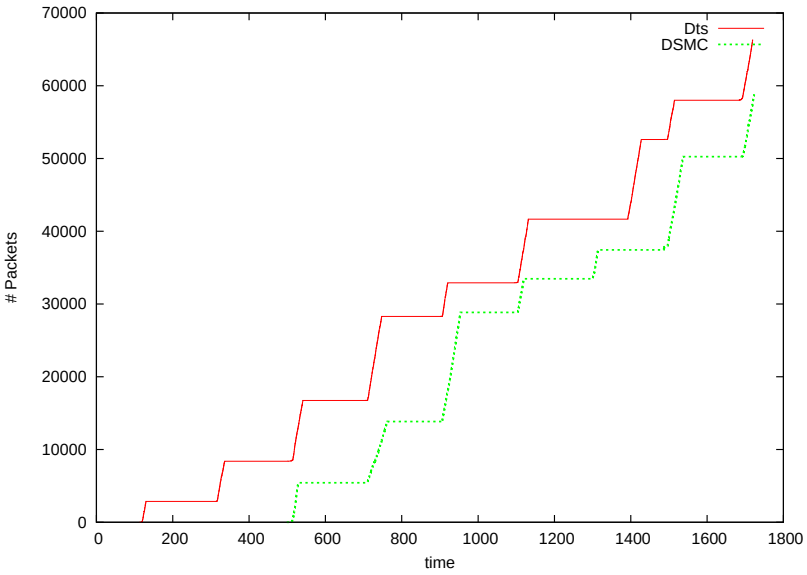

Figure A.61: Single Run Scenario 1 Configuration 11, Seed 1

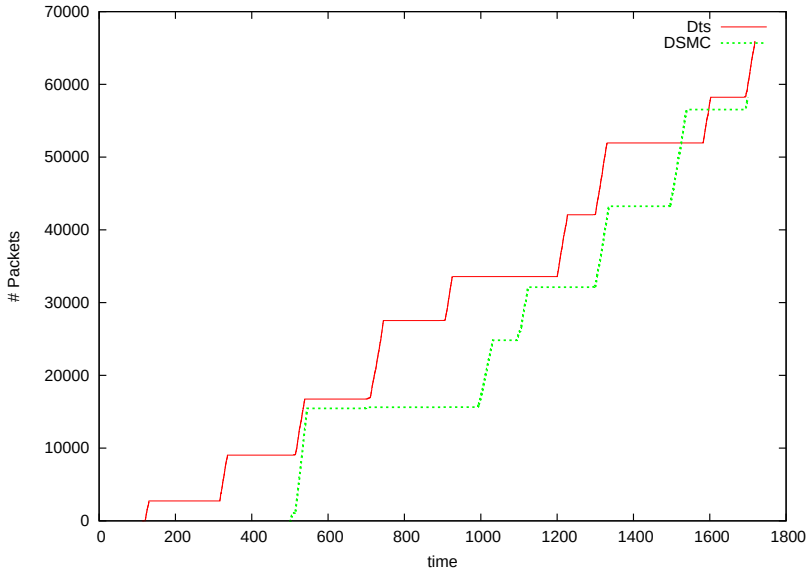

Figure A.62: Single Run Scenario 1 Configuration 11, Seed 2

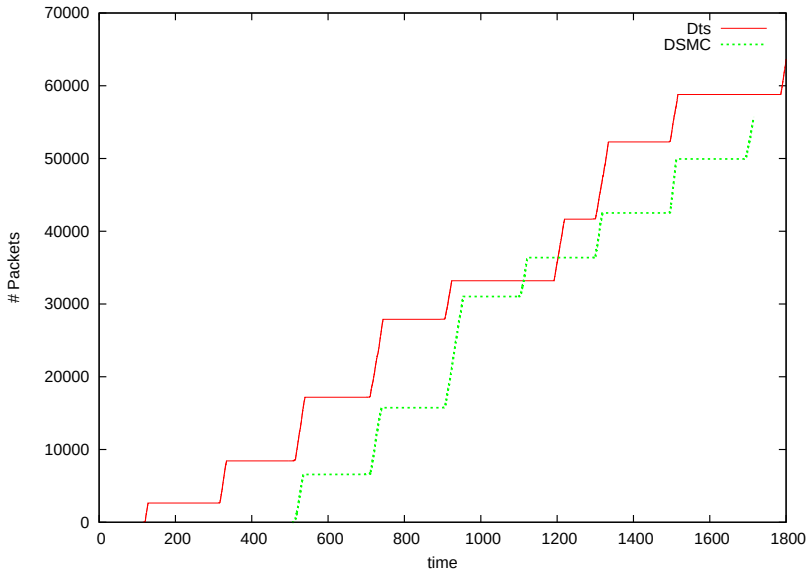

Figure A.63: Single Run Scenario 1 Configuration 11, Seed 3

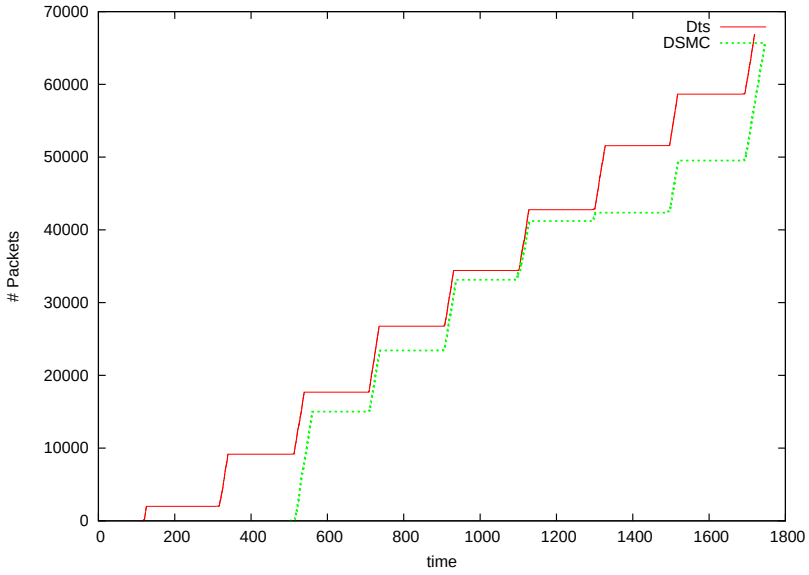

Figure A.64: Single Run Scenario 1 Configuration 11, Seed 4

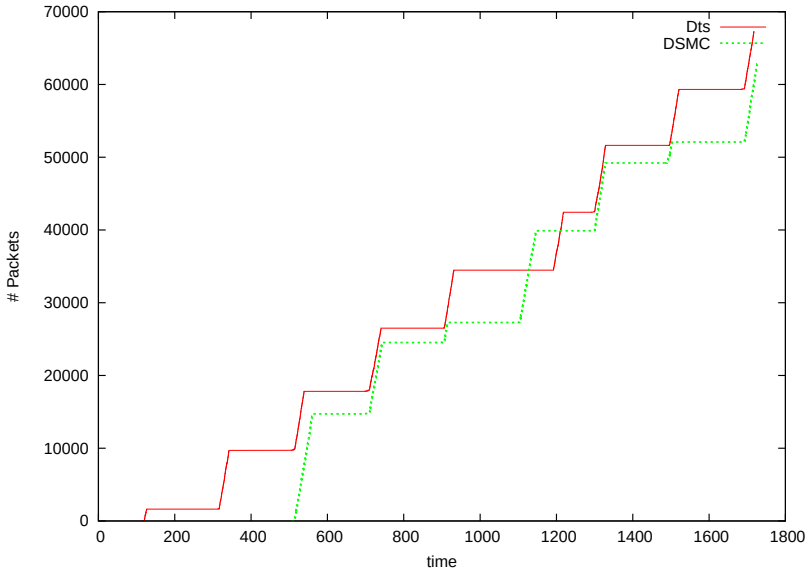

Figure A.65: Single Run Scenario 1 Configuration 11, Seed 5

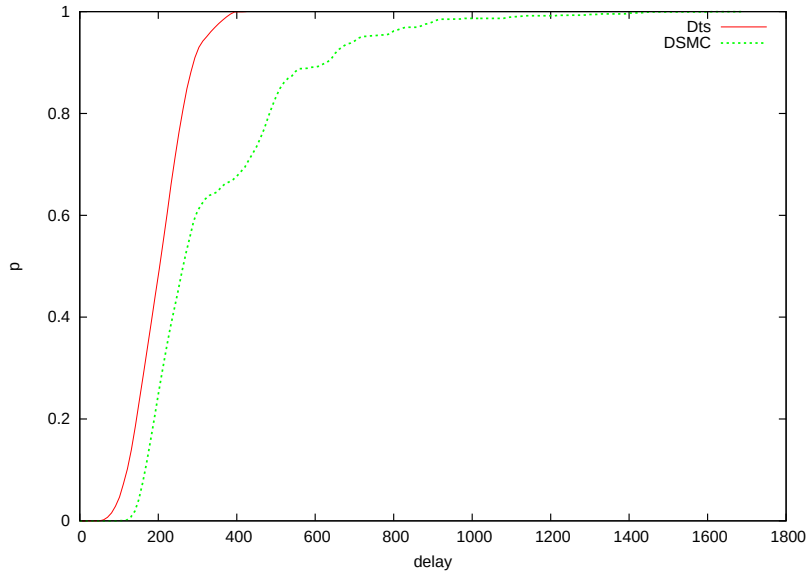

Figure A.66: CDF for All Runs With Configuration 11 in Scenario 1

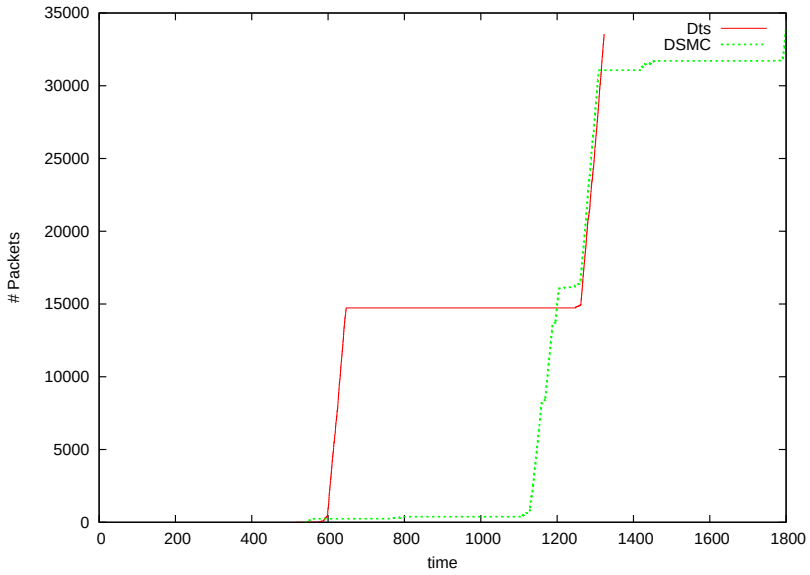

Figure A.67: Single Run Scenario 1 Configuration 12, Seed 1

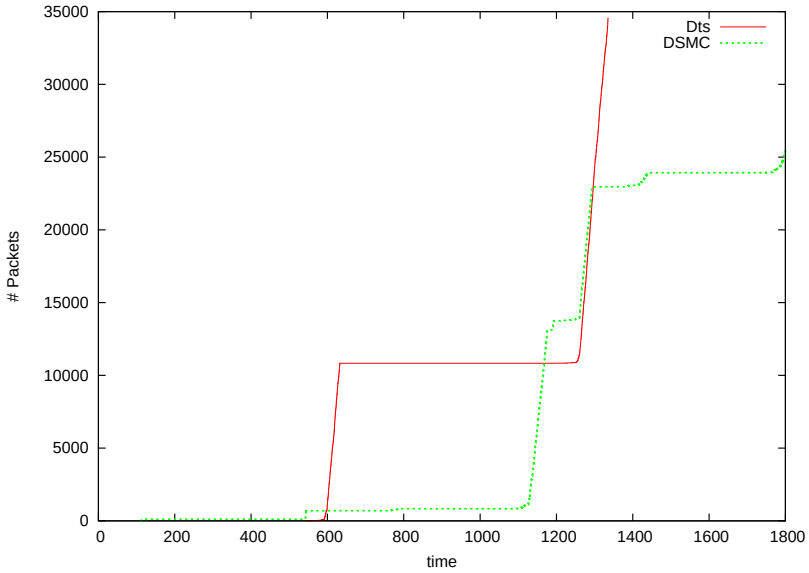

Figure A.68: Single Run Scenario 1 Configuration 12, Seed 2

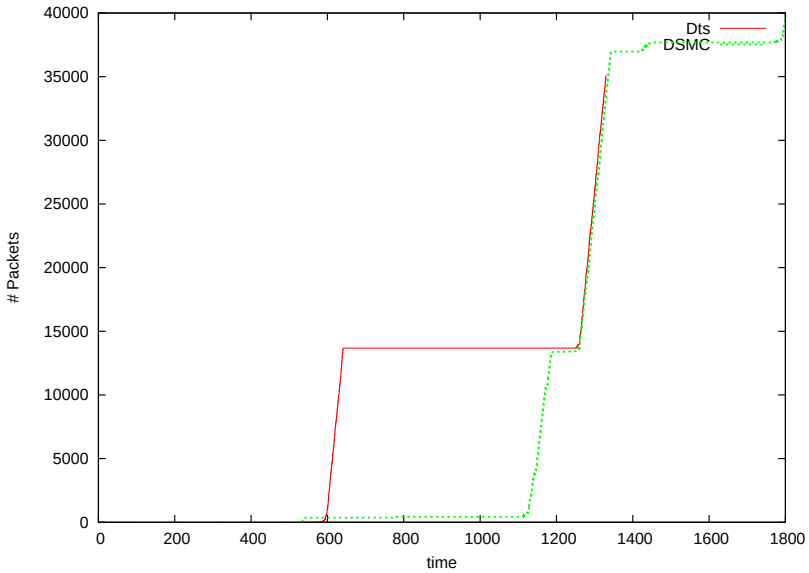

Figure A.69: Single Run Scenario 1 Configuration 12, Seed 3

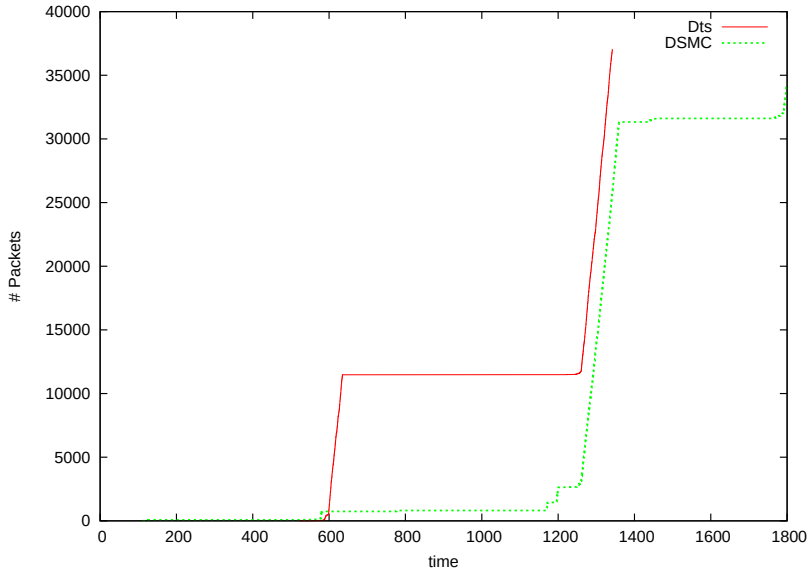

Figure A.70: Single Run Scenario 1 Configuration 12, Seed 4

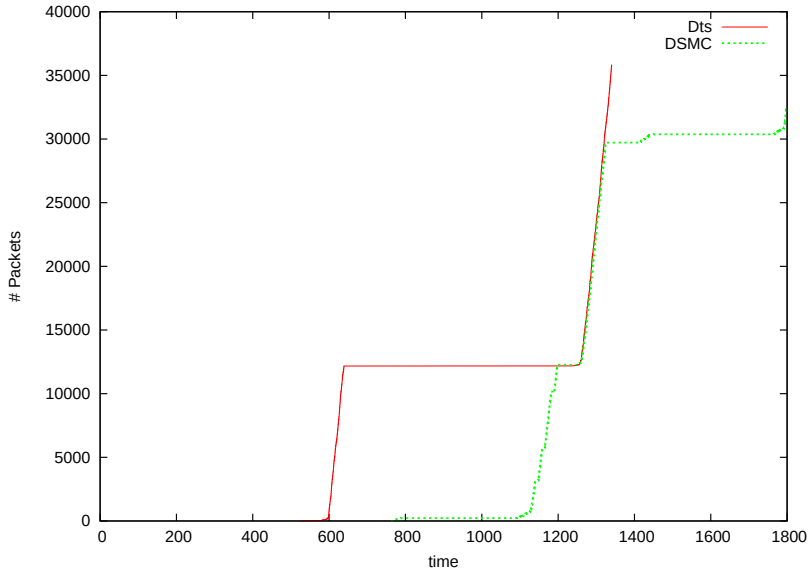

Figure A.71: Single Run Scenario 1 Configuration 12, Seed 5

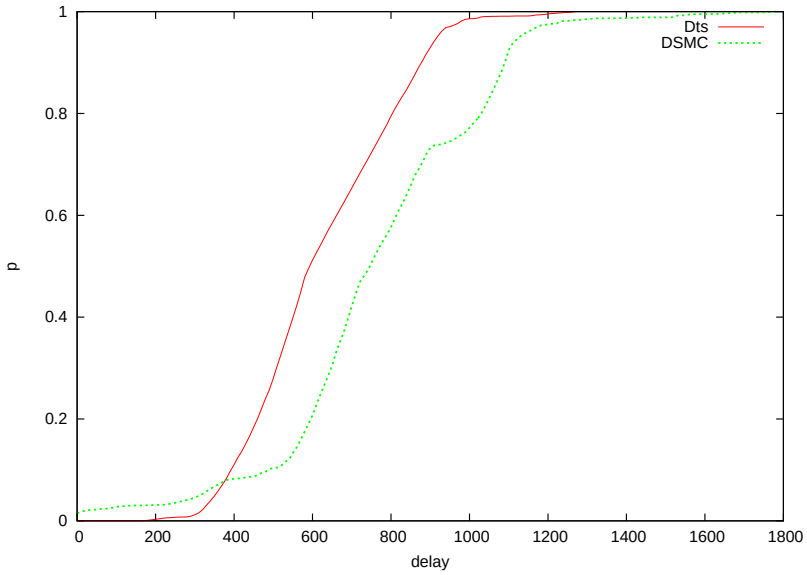

Figure A.72: CDF for All Runs With Configuration 12 in Scenario 1

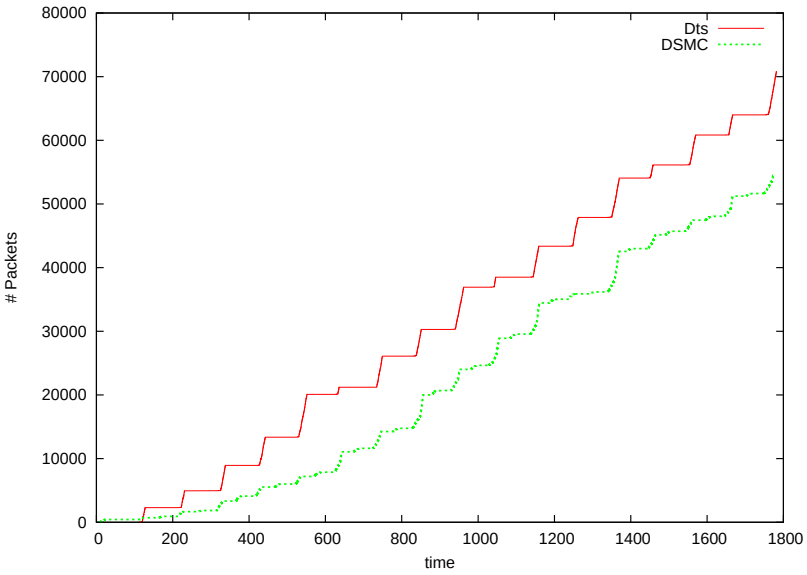

Figure A.73: Single Run Scenario 1 Configuration 13, Seed 1

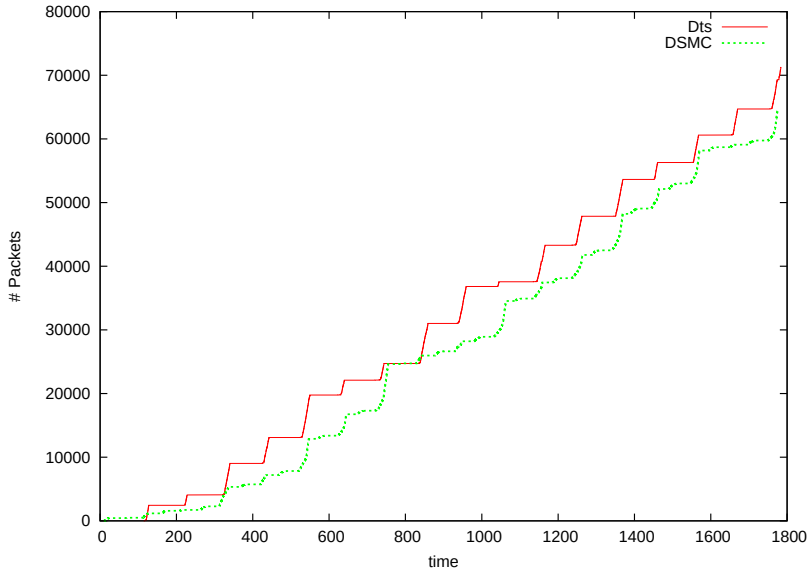

Figure A.74: Single Run Scenario 1 Configuration 13, Seed 2

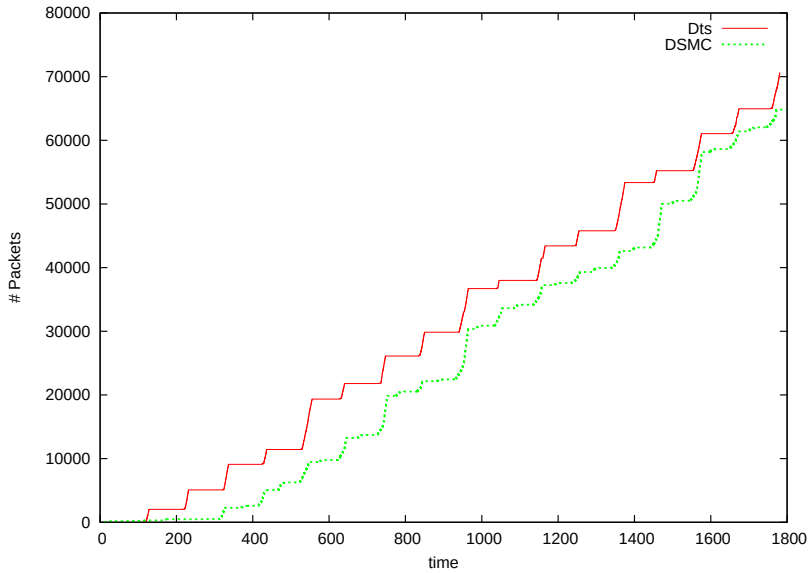

Figure A.75: Single Run Scenario 1 Configuration 13, Seed 3

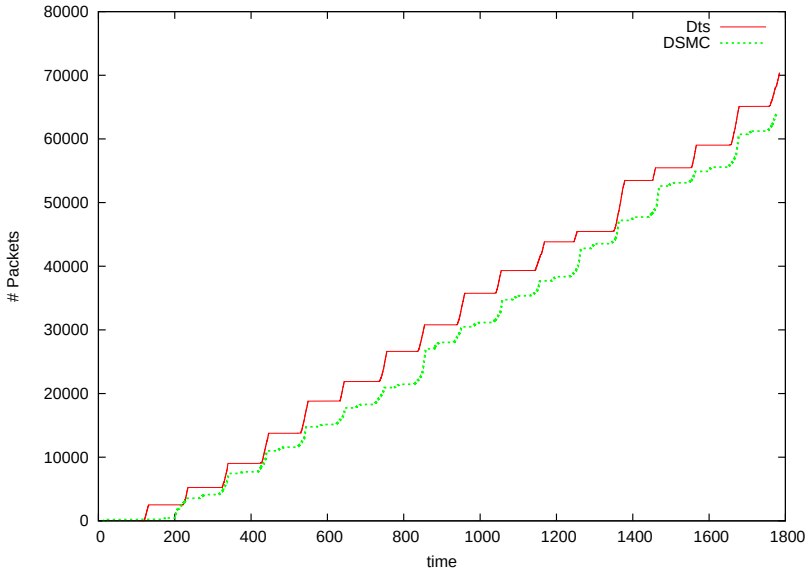

Figure A.76: Single Run Scenario 1 Configuration 13, Seed 4

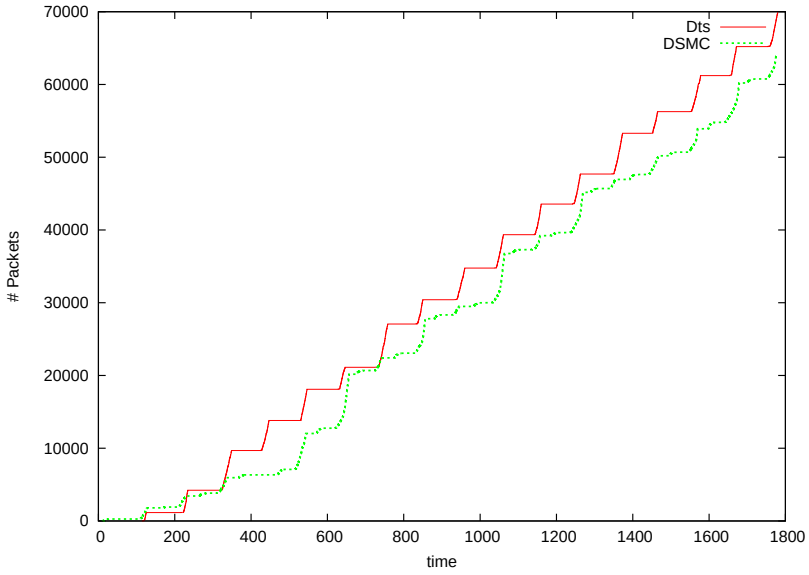

Figure A.77: Single Run Scenario 1 Configuration 13, Seed 5

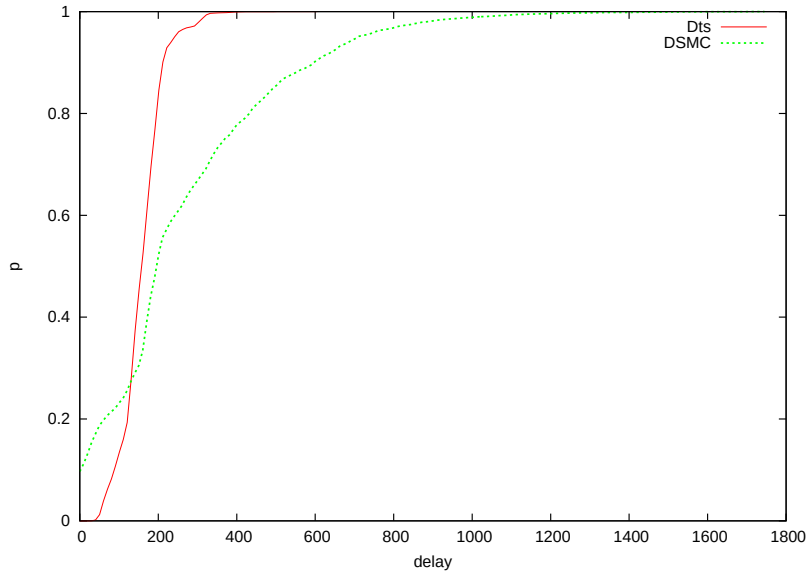

Figure A.78: CDF for All Runs With Configuration 13 in Scenario 1

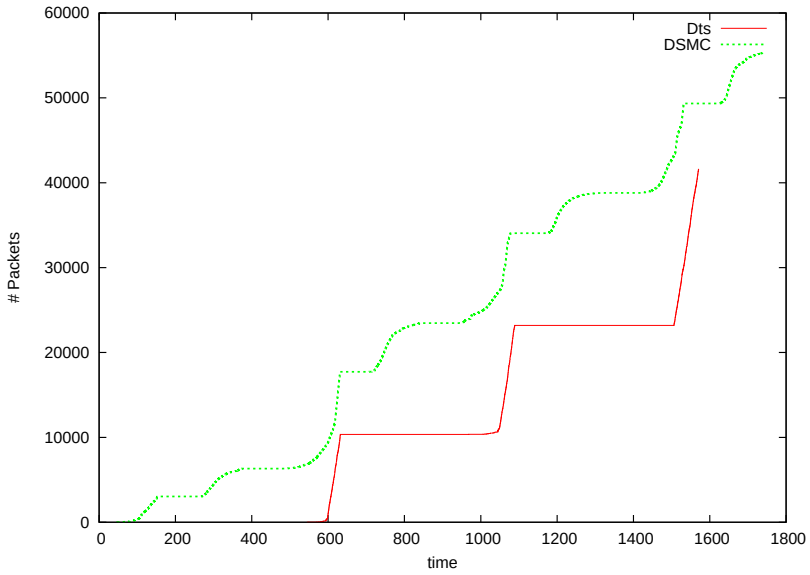

Figure A.79: Single Run Scenario 1 Configuration 14, Seed 1

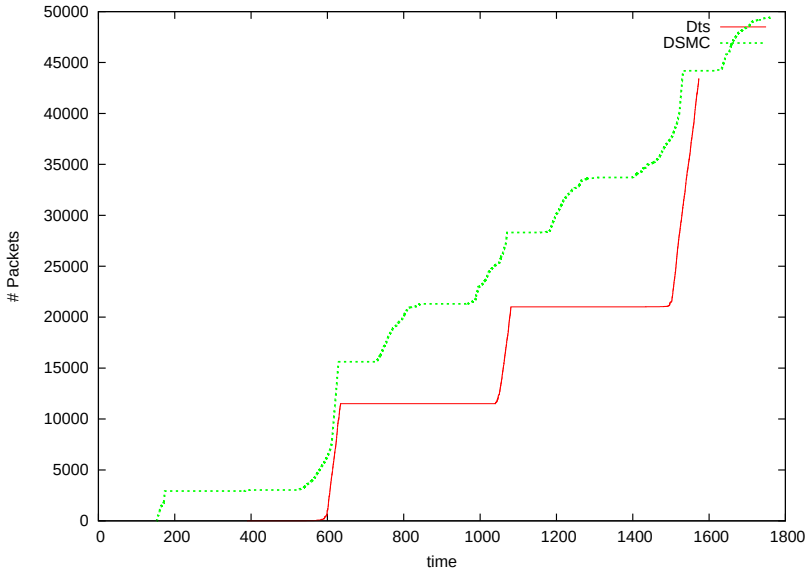

Figure A.80: Single Run Scenario 1 Configuration 14, Seed 2

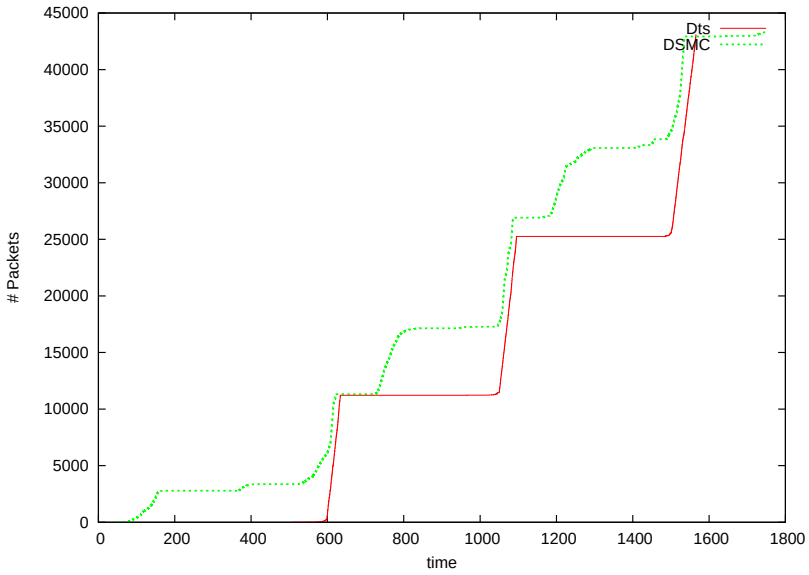

Figure A.81: Single Run Scenario 1 Configuration 14, Seed 3

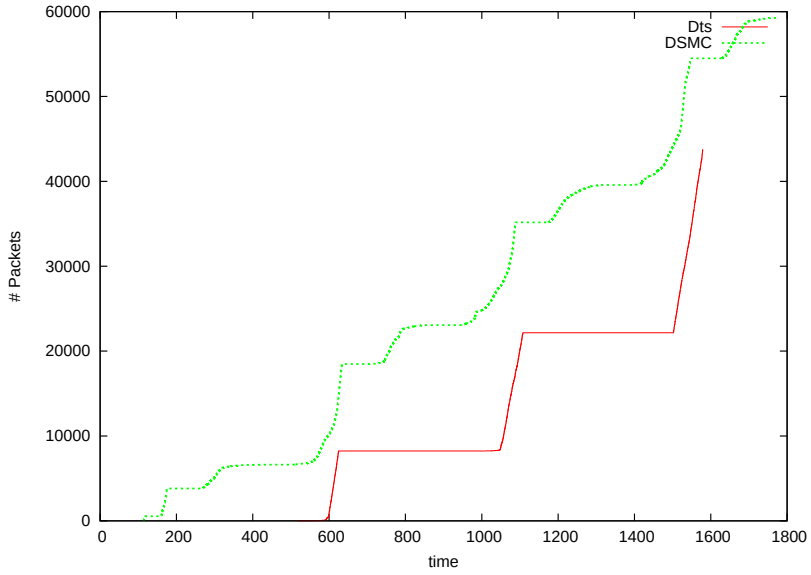

Figure A.82: Single Run Scenario 1 Configuration 14, Seed 4

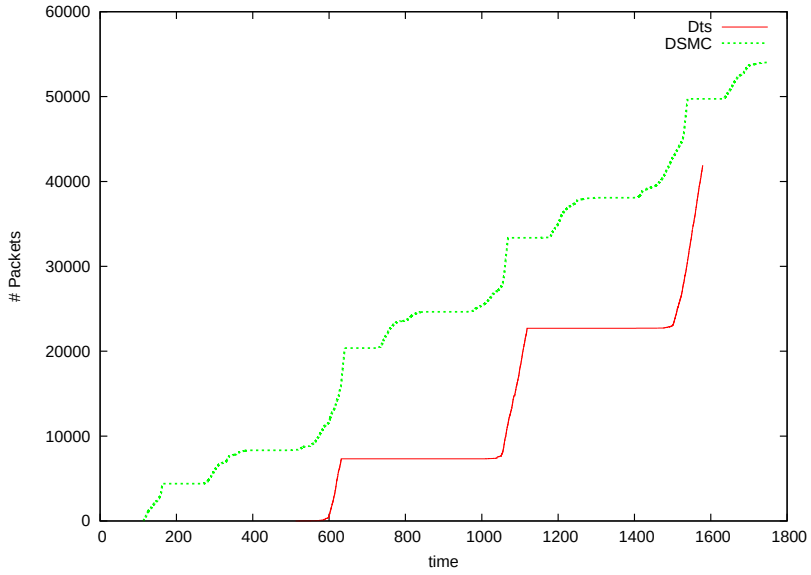

Figure A.83: Single Run Scenario 1 Configuration 14, Seed 5

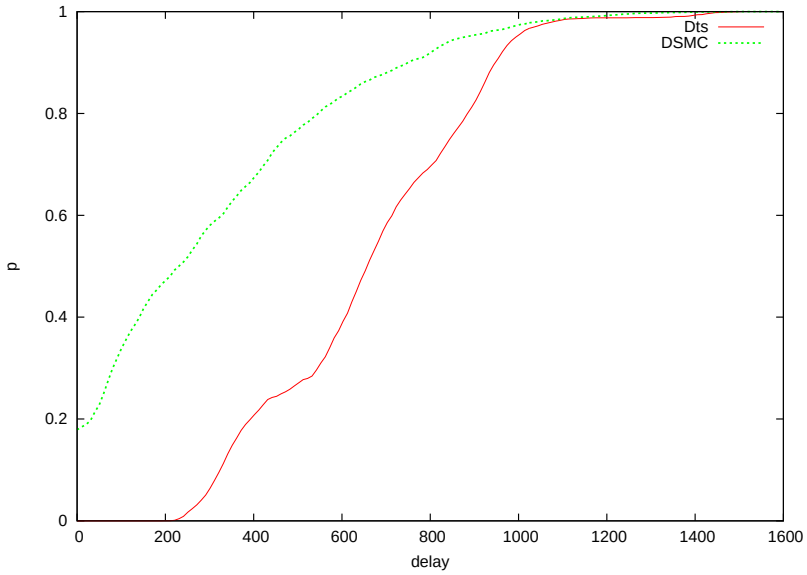

Figure A.84: CDF for All Runs With Configuration 14 in Scenario 1

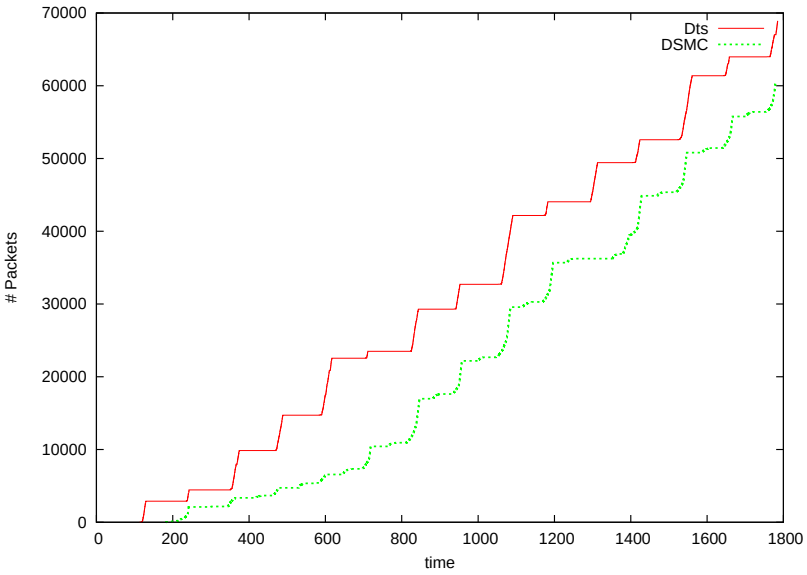

Figure A.85: Single Run Scenario 1 Configuration 15, Seed 1

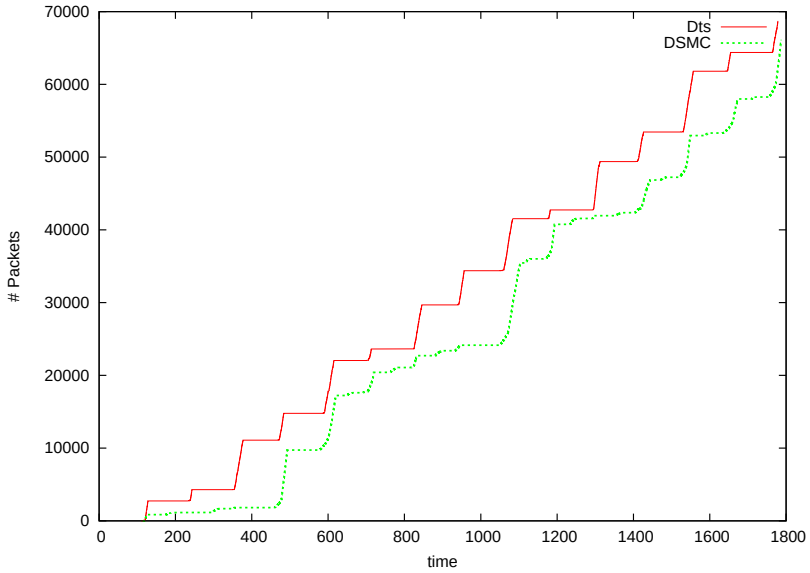

Figure A.86: Single Run Scenario 1 Configuration 15, Seed 2

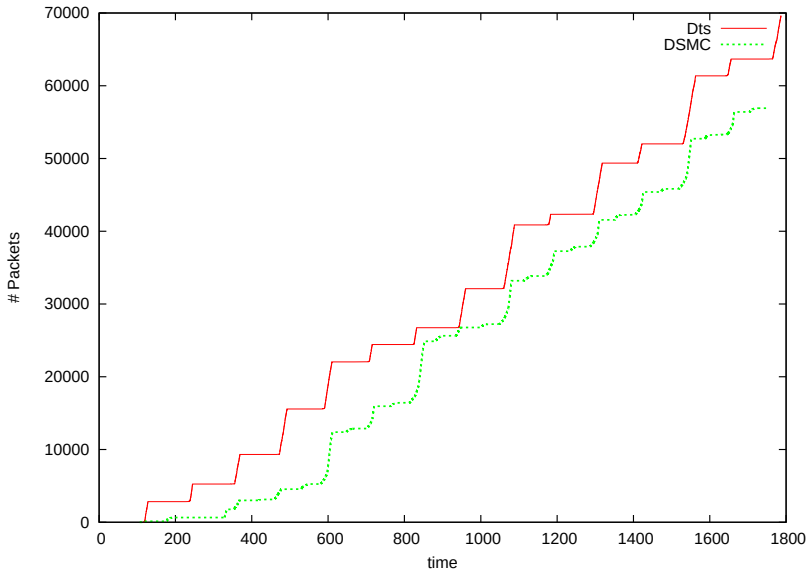

Figure A.87: Single Run Scenario 1 Configuration 15, Seed 3

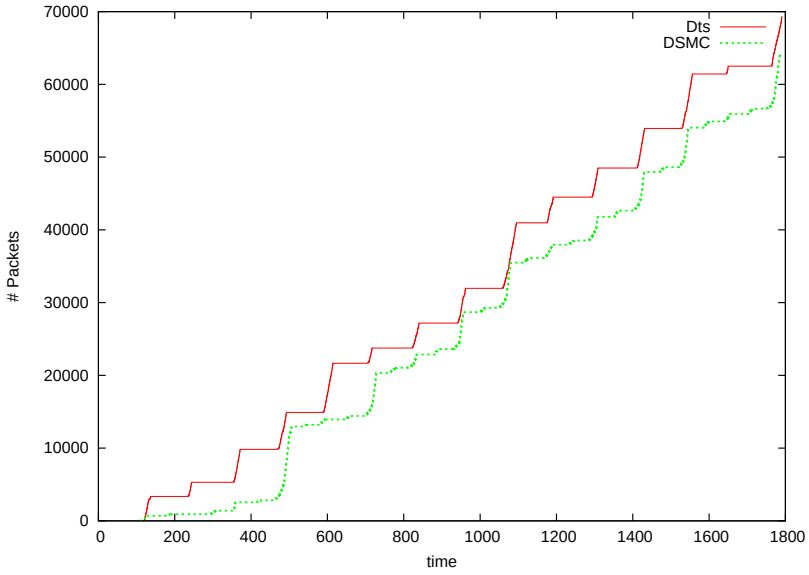

Figure A.88: Single Run Scenario 1 Configuration 15, Seed 4

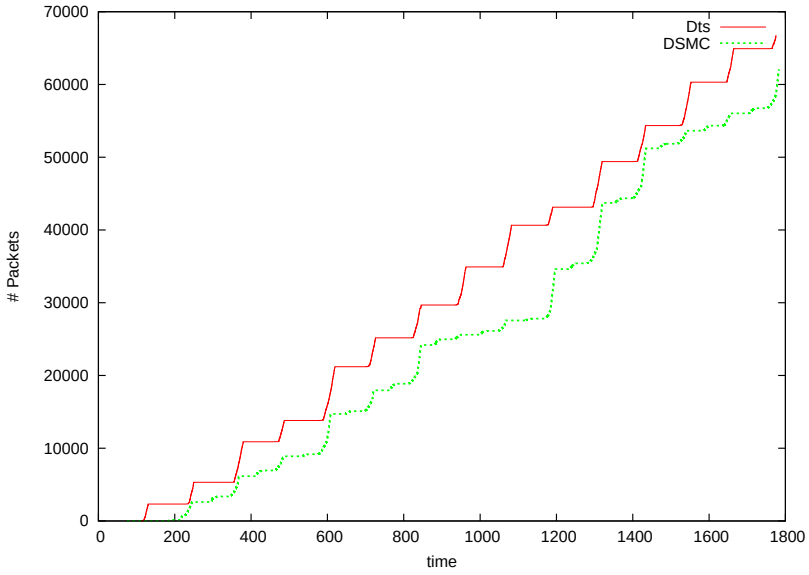

Figure A.89: Single Run Scenario 1 Configuration 15, Seed 5

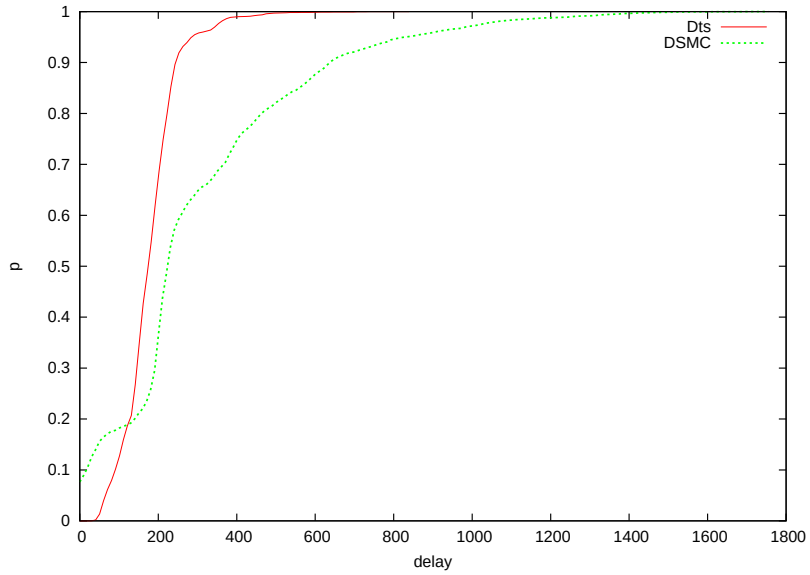

Figure A.90: CDF for All Runs With Configuration 15 in Scenario 1

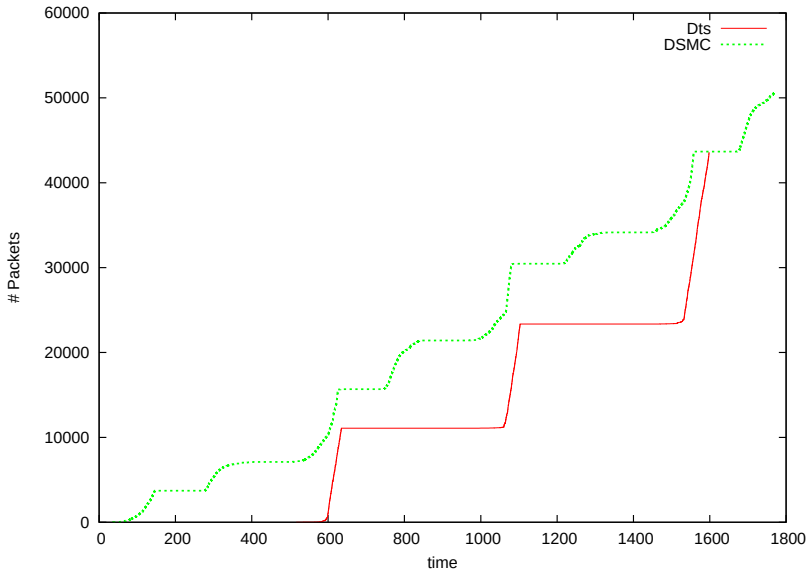

Figure A.91: Single Run Scenario 1 Configuration 16, Seed 1

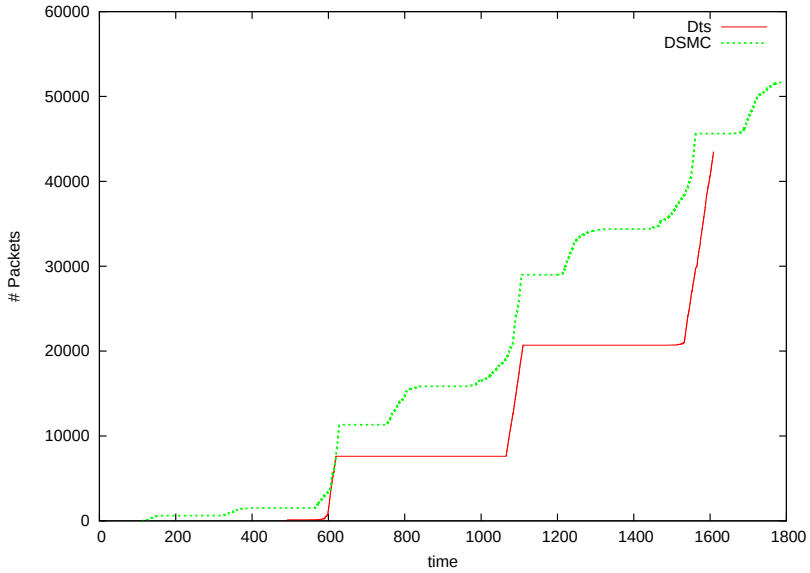

Figure A.92: Single Run Scenario 1 Configuration 16, Seed 2

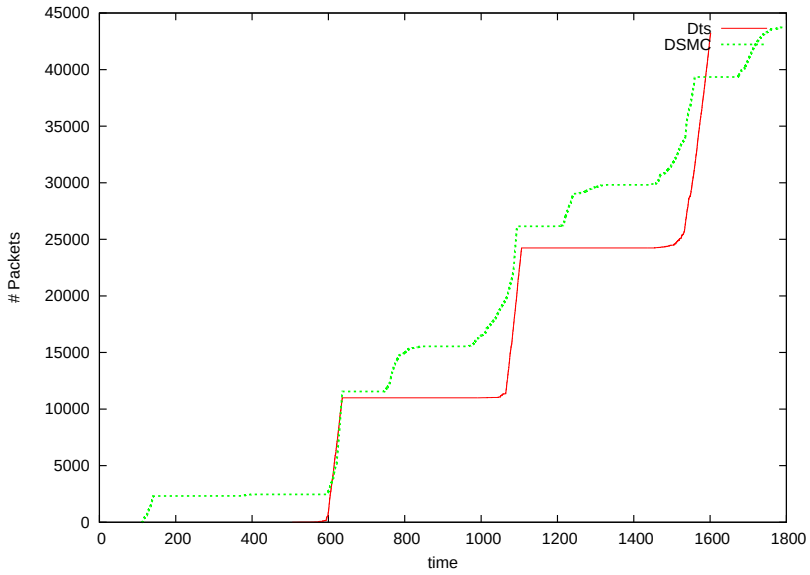

Figure A.93: Single Run Scenario 1 Configuration 16, Seed 3

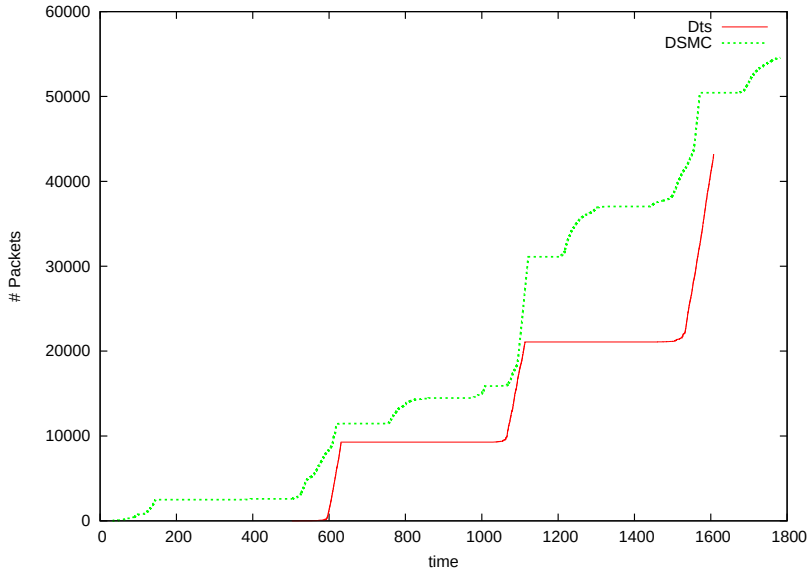

Figure A.94: Single Run Scenario 1 Configuration 16, Seed 4

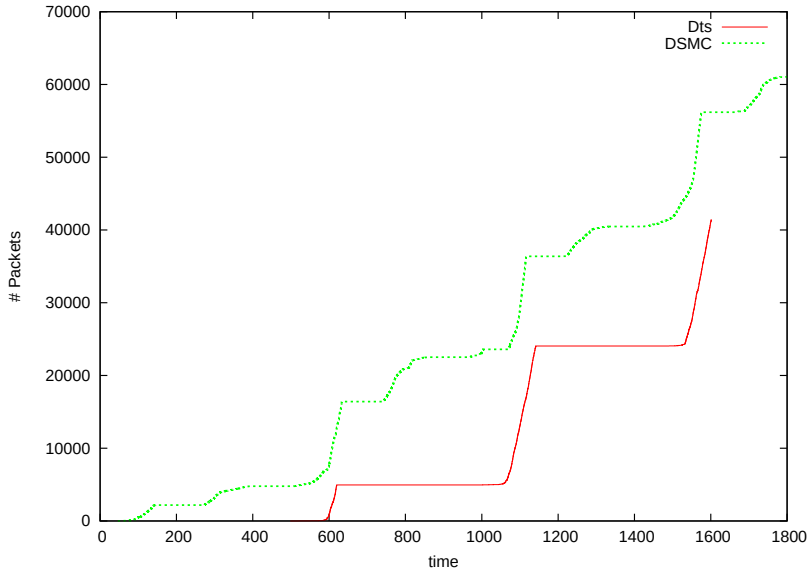

Figure A.95: Single Run Scenario 1 Configuration 16, Seed 5

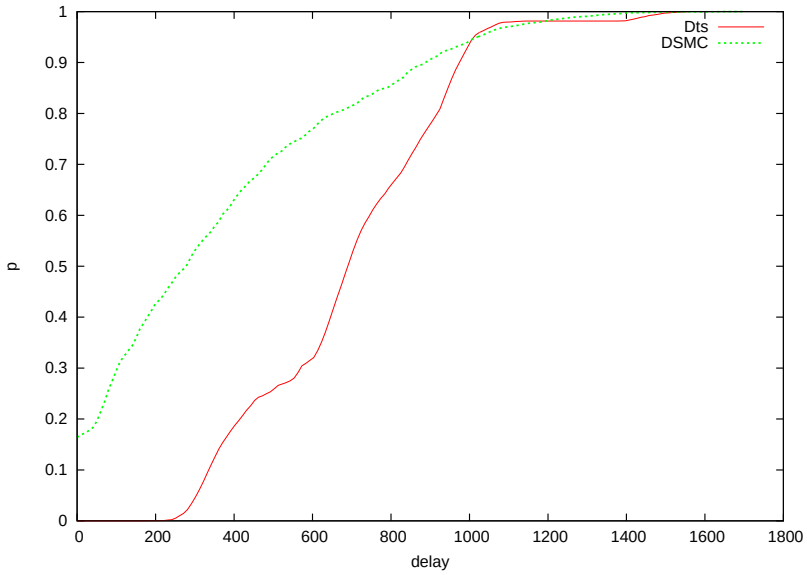

Figure A.96: CDF for All Runs With Configuration 16 in Scenario 1

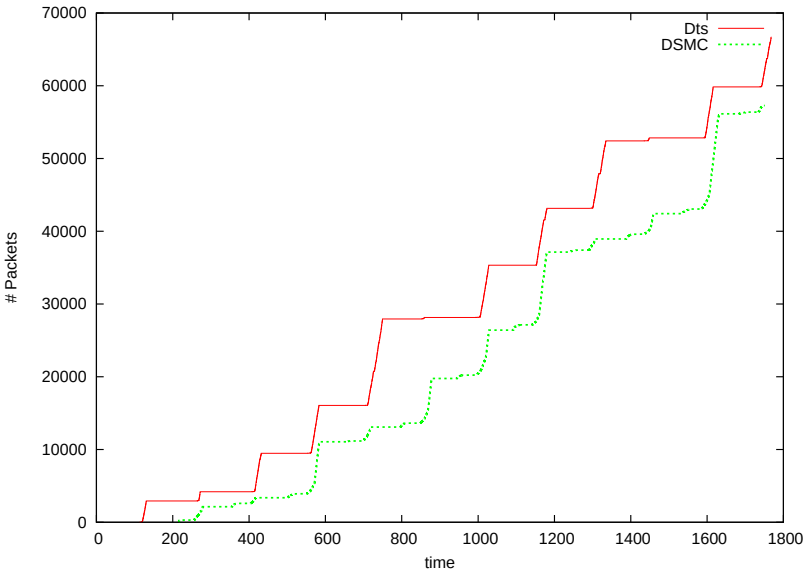

Figure A.97: Single Run Scenario 1 Configuration 17, Seed 1

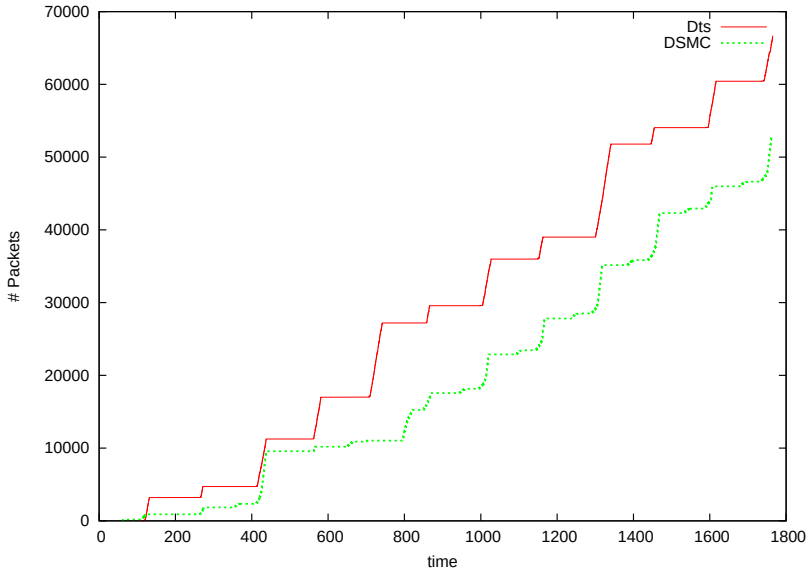

Figure A.98: Single Run Scenario 1 Configuration 17, Seed 2

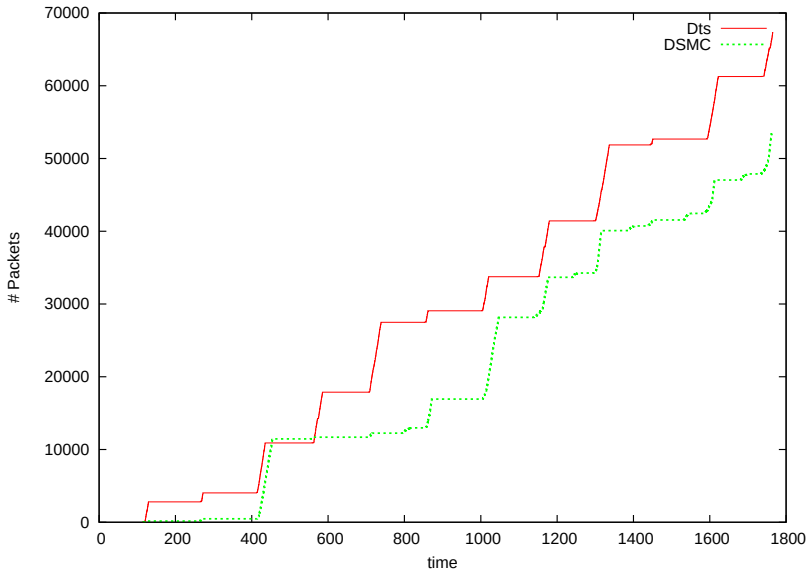

Figure A.99: Single Run Scenario 1 Configuration 17, Seed 3

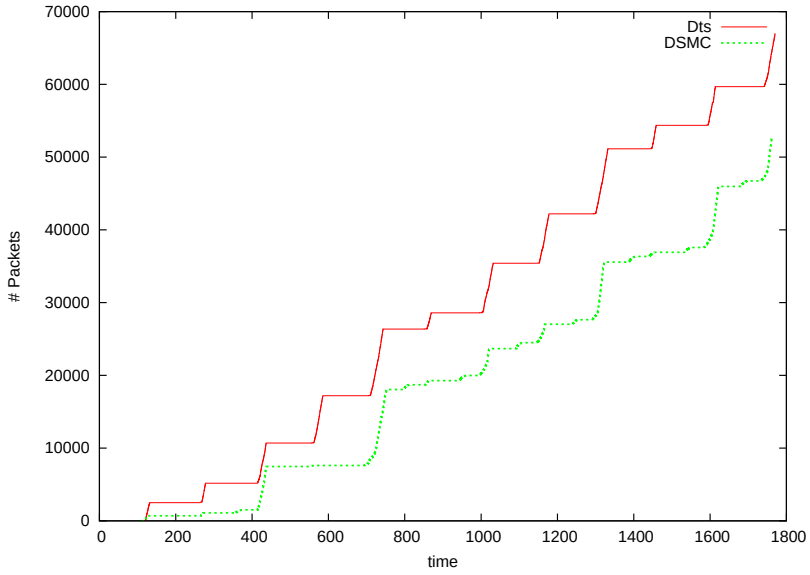

Figure A.100: Single Run Scenario 1 Configuration 17, Seed 4

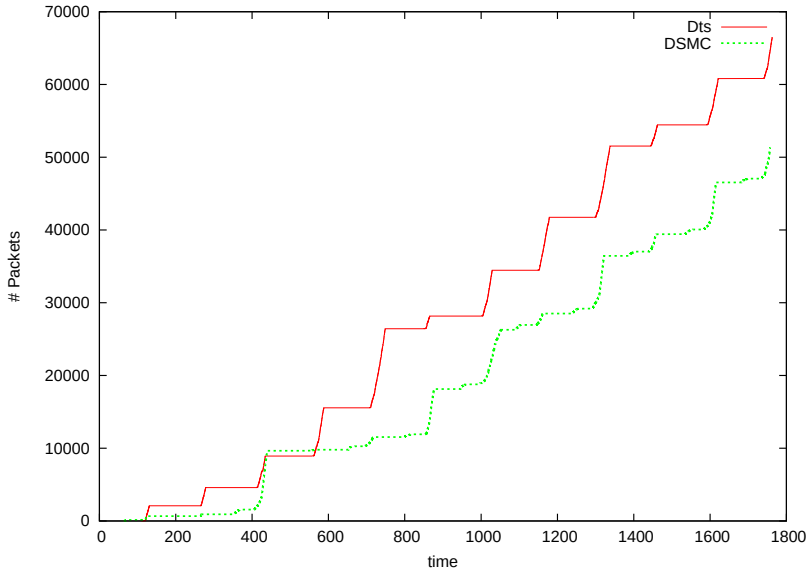

Figure A.101: Single Run Scenario 1 Configuration 17, Seed 5

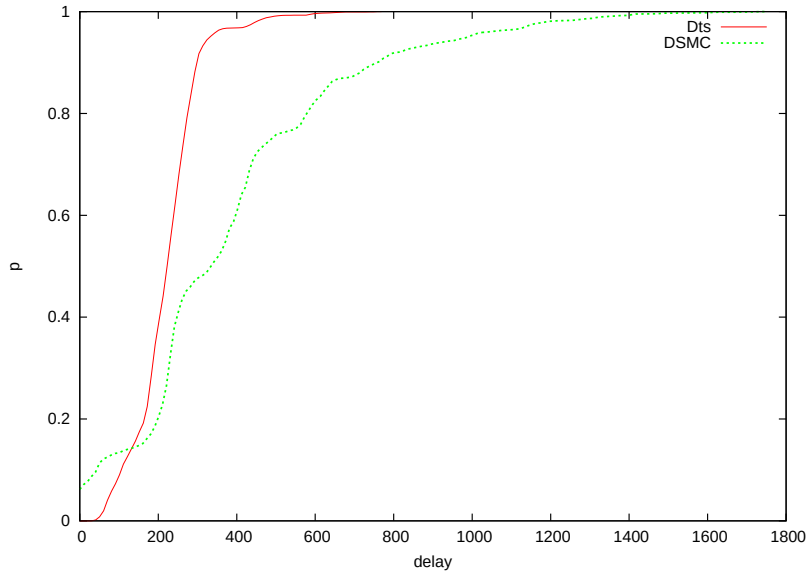

Figure A.102: CDF for All Runs With Configuration 17 in Scenario 1

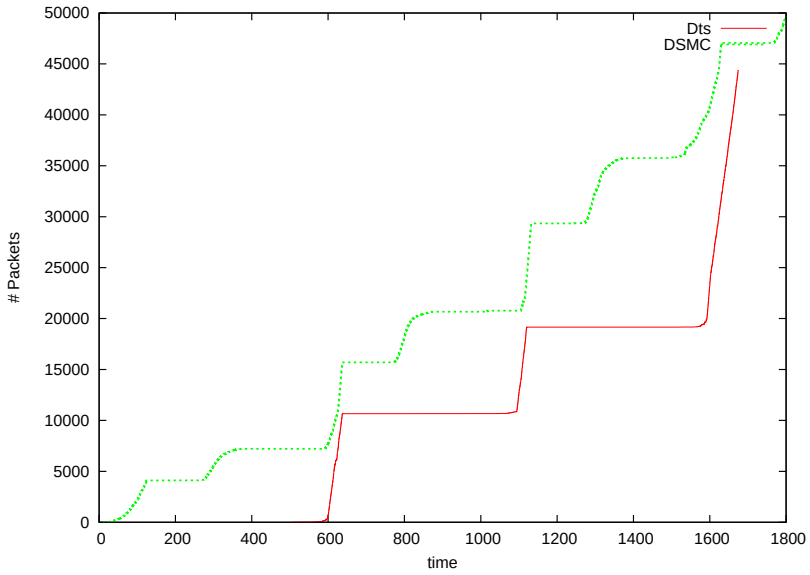

Figure A.103: Single Run Scenario 1 Configuration 18, Seed 1

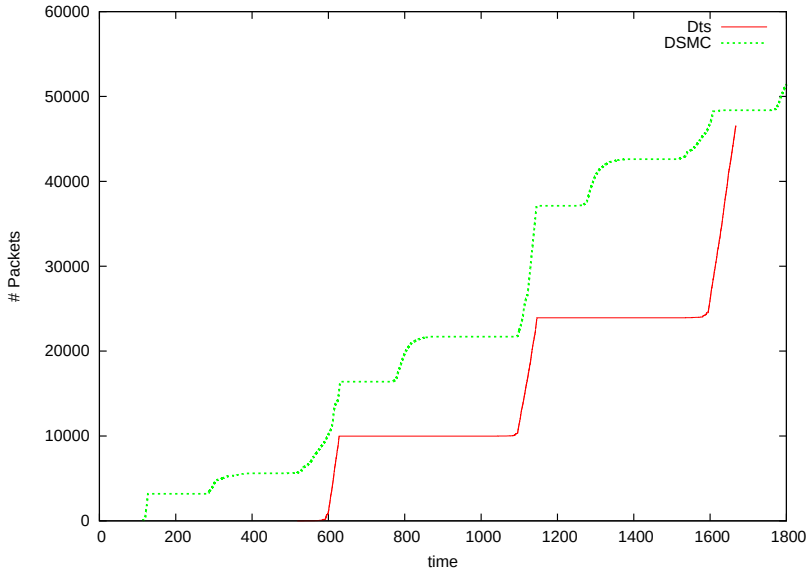

Figure A.104: Single Run Scenario 1 Configuration 18, Seed 2

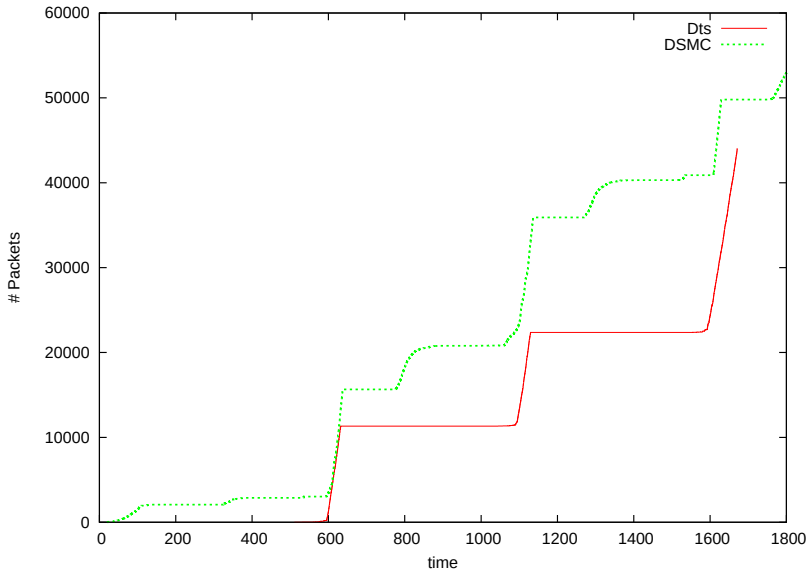

Figure A.105: Single Run Scenario 1 Configuration 18, Seed 3

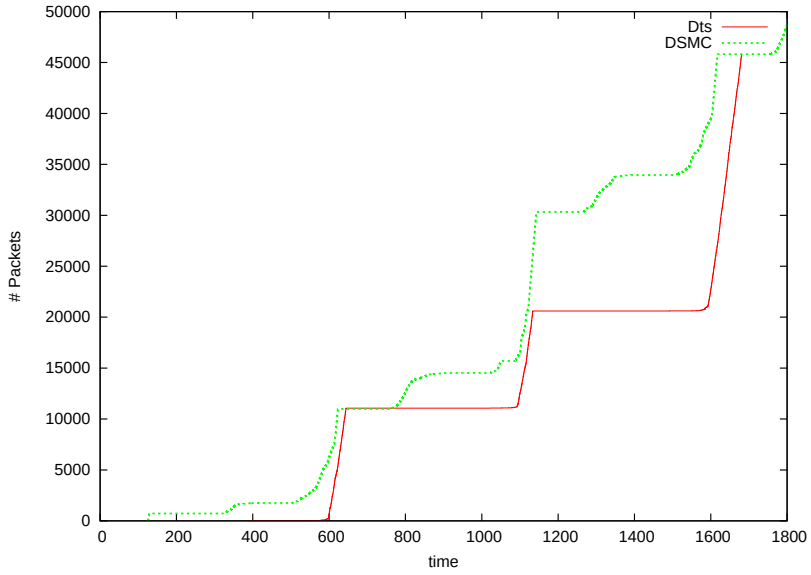

Figure A.106: Single Run Scenario 1 Configuration 18, Seed 4

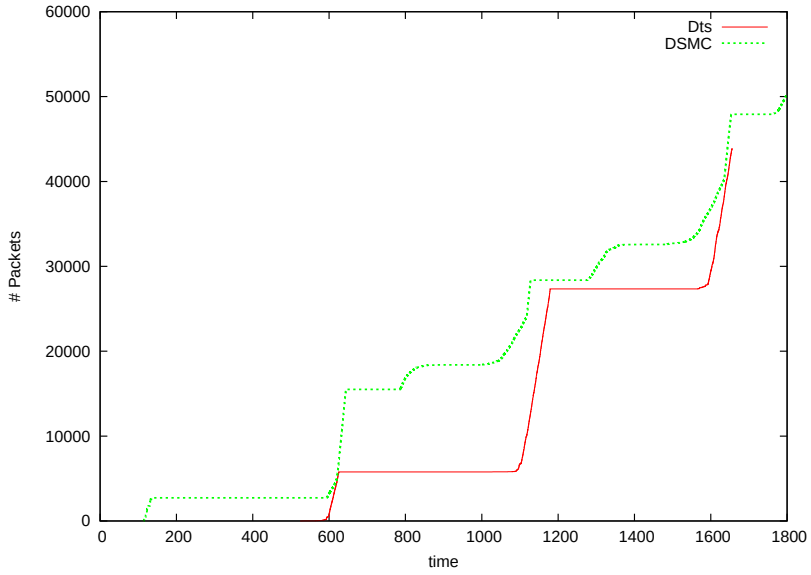

Figure A.107: Single Run Scenario 1 Configuration 18, Seed 5

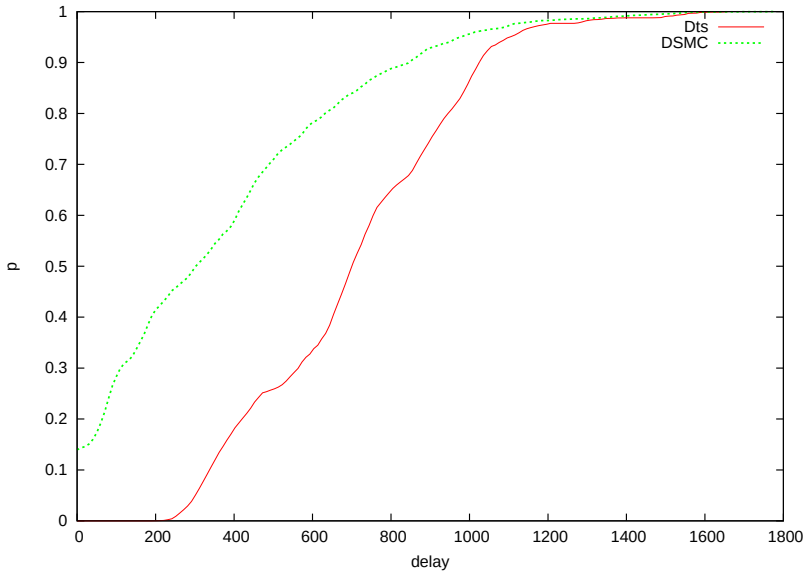

Figure A.108: CDF for All Runs With Configuration 18 in Scenario 1

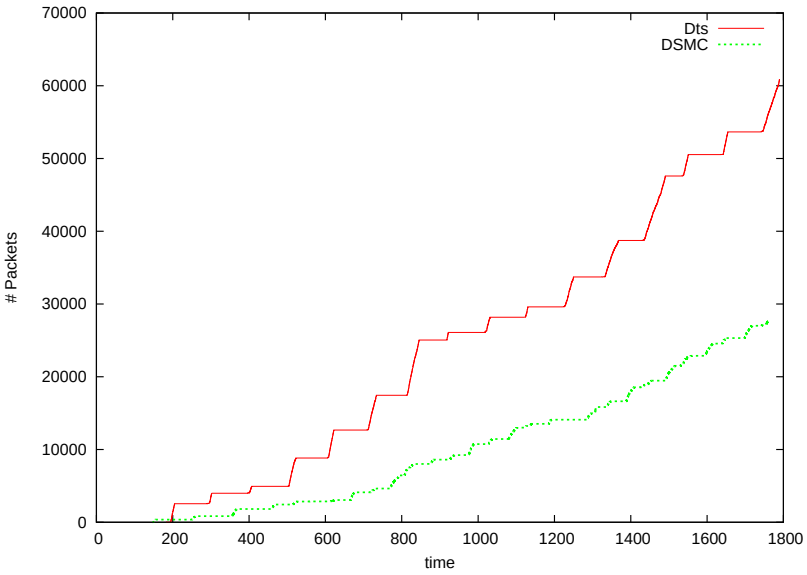

Figure A.109: Single Run Scenario 2 Configuration 1, Seed 1

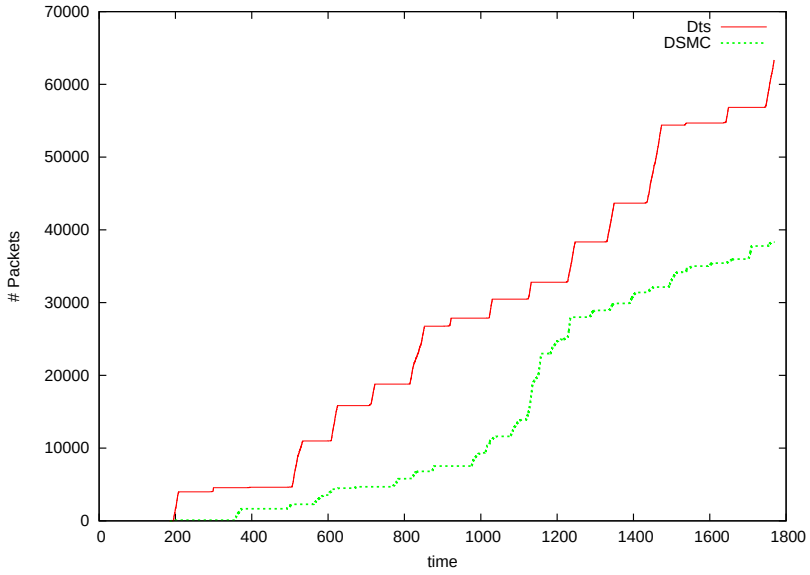

Figure A.110: Single Run Scenario 2 Configuration 1, Seed 2

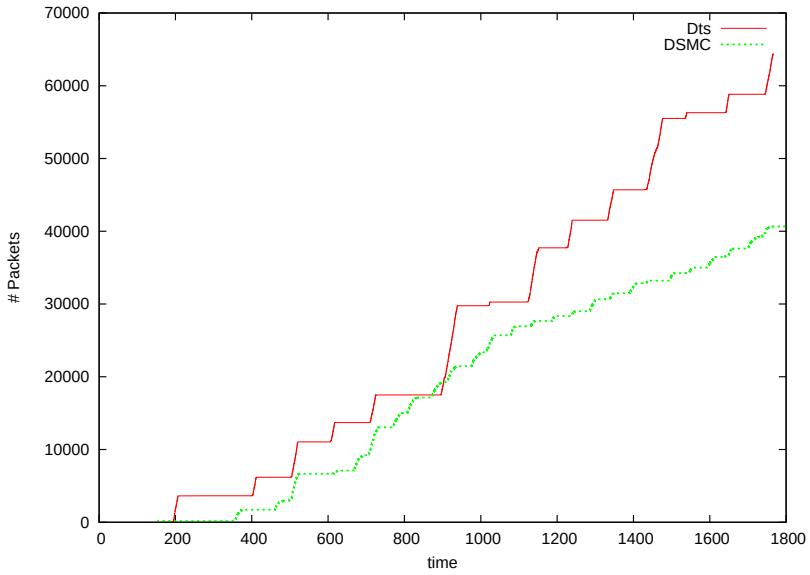

Figure A.111: Single Run Scenario 2 Configuration 1, Seed 3

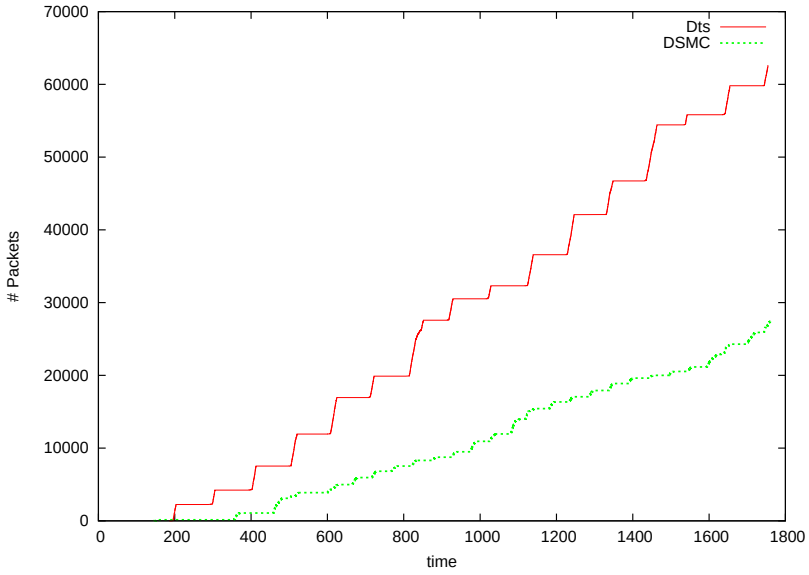

Figure A.112: Single Run Scenario 2 Configuration 1, Seed 4

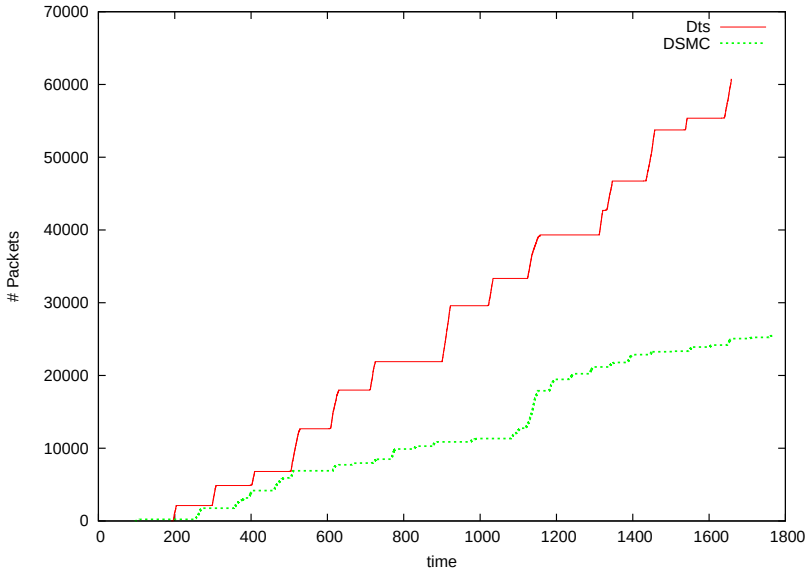

Figure A.113: Single Run Scenario 2 Configuration 1, Seed 5

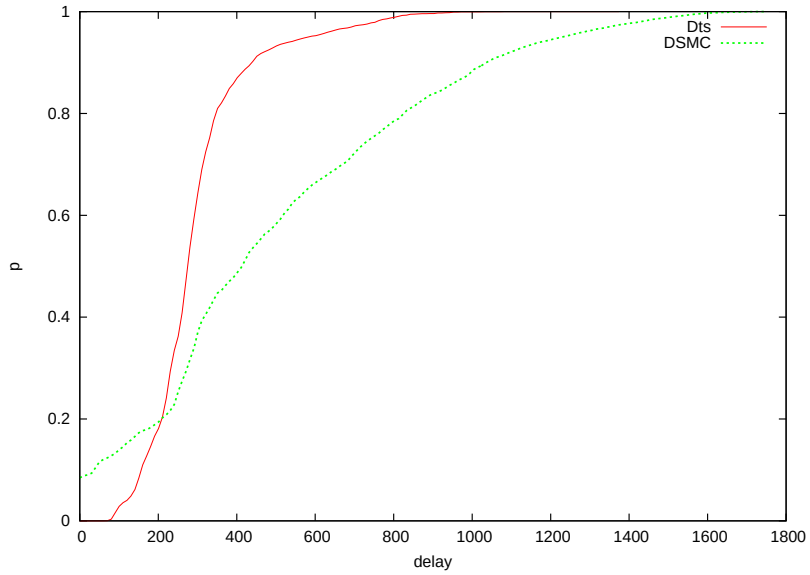

Figure A.114: CDF for All Runs With Configuration 1 in Scenario 2

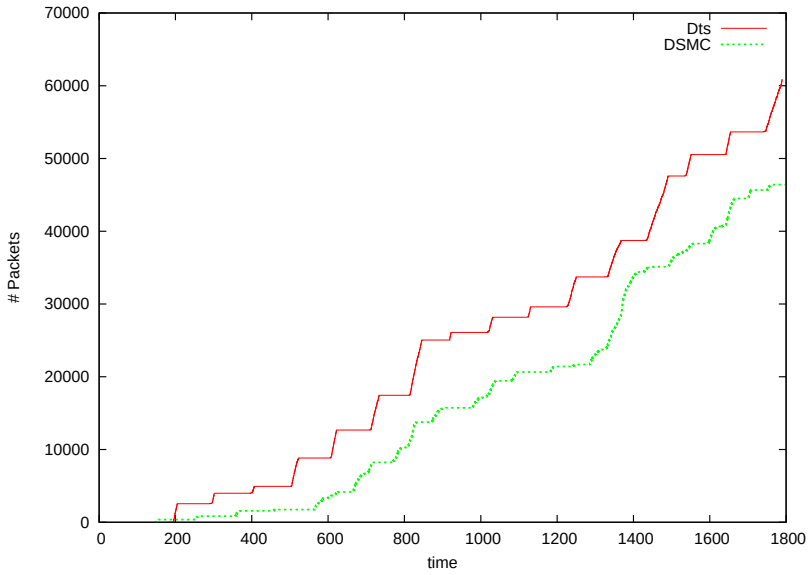

Figure A.115: Single Run Scenario 2 Configuration 2, Seed 1

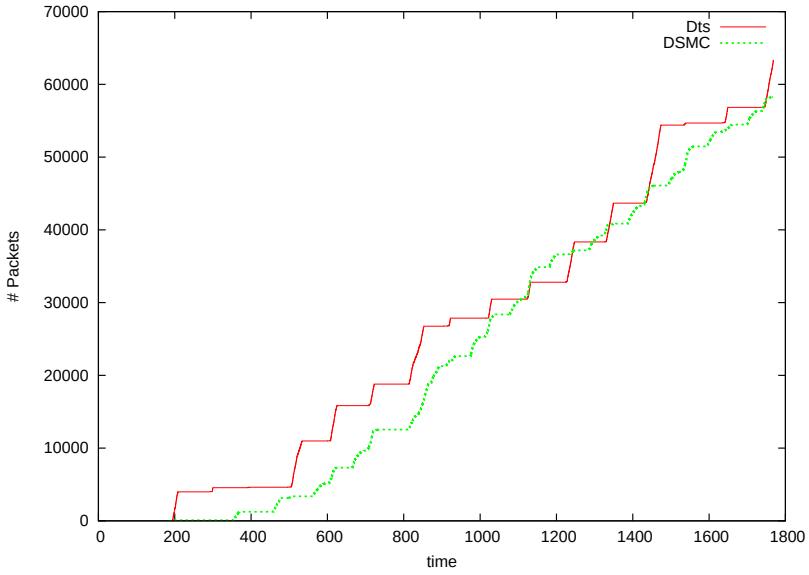

Figure A.116: Single Run Scenario 2 Configuration 2, Seed 2

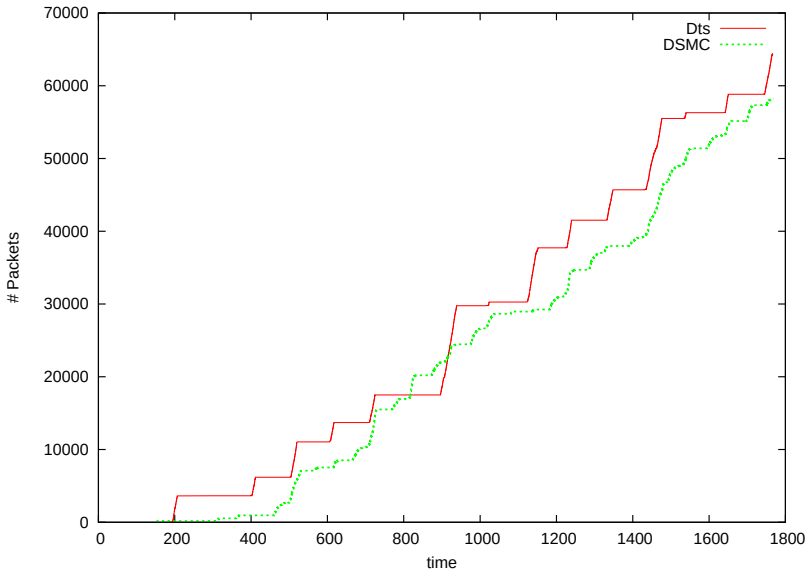

Figure A.117: Single Run Scenario 2 Configuration 2, Seed 3

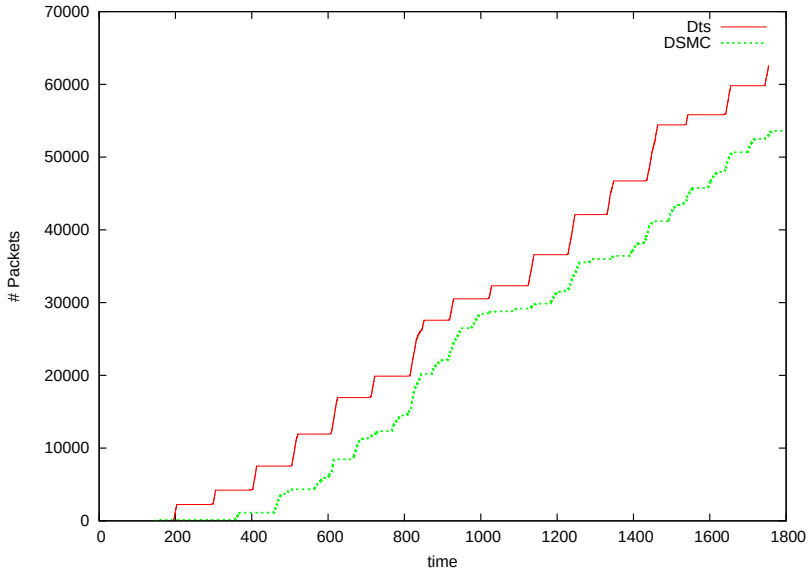

Figure A.118: Single Run Scenario 2 Configuration 2, Seed 4

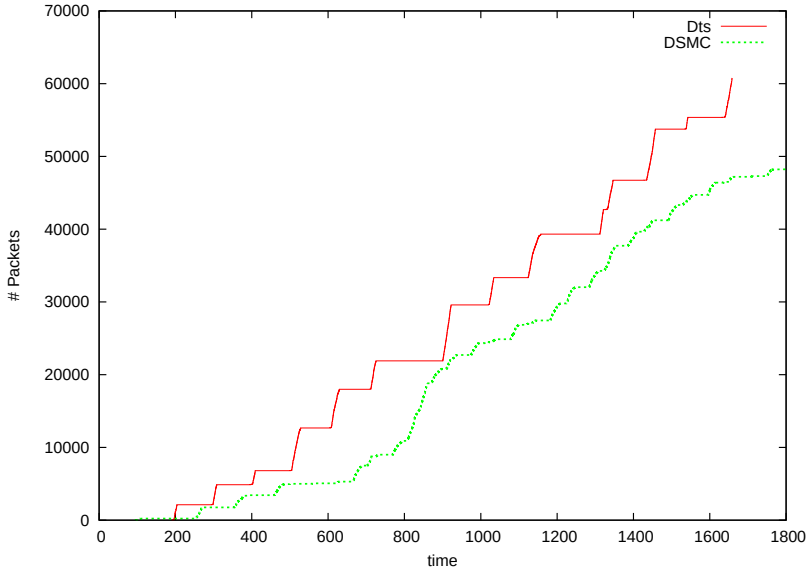

Figure A.119: Single Run Scenario 2 Configuration 2, Seed 5

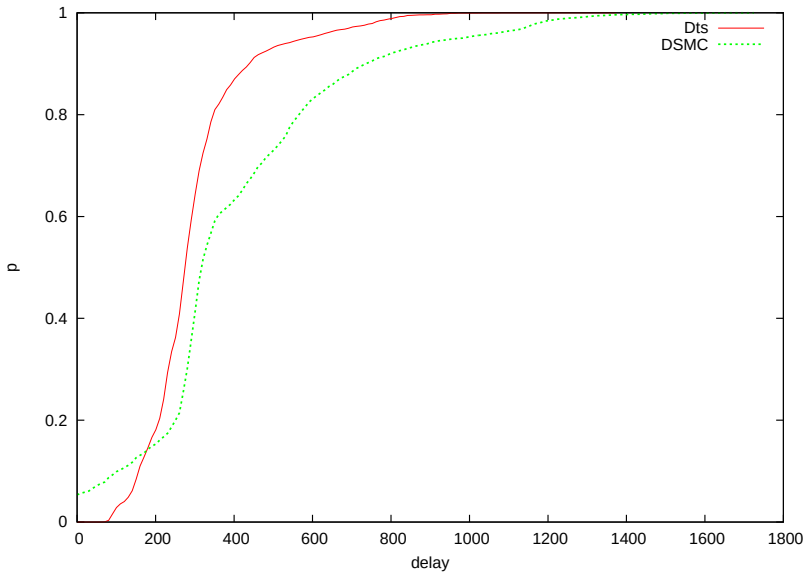

Figure A.120: CDF for All Runs With Configuration 2 in Scenario 2

Appendix B

Code

In this appendix, we display the most important implementation code for DSMC and the part of the Dts-Overlay that interacts with DSMC. In Listing B.1, we show the decision maker, which makes routing decision for the Dts-Overlay. It interacts with the resource manager (B.2) to find routes and the carrier manager (Listing B.3) to find carrier nodes. For DSMC, the carrier manager interacts with the context manager (Listing B.4) to find the carrier node with the highest delivery probability to a given destination. A detailed explanation of how the Dts-Overlay and DSMC interacts is given in Chapter 6.

Listing B.1: Code for Decision Maker

```
/* -*- Mode:C++; c-file-style:"gnu"; indent-tabs-mode:nil; -*- */
/*
 * Copyright (c) Stein Kristiansen and Morten Lindeberg 2010
 *
 * This program is free software; you can redistribute it and/or modify
 * it under the terms of the GNU General Public License version 2 as
 * published by the Free Software Foundation;
 *
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with this program; if not, write to the Free Software
 * Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA
 *
 * Author: Stein Kristiansen (steikr@ifi.uio.no)
 *         Morten Lindeberg (mglindeb@ifi.uio.no)
 *         Jan Erik Haavet (janehaa@ifi.uio.no)
 */
#include <iostream>
#include "ns3/ipv4-address.h"
#include "ns3/ipv4.h"
```

```
#include "ns3/simulator.h"
#include "ns3/double.h"
#include "ns3/uinteger.h"
#include "decision-maker.h"
#include "overlaymessage.h"
#include "dts-overlay.h"
#include "carrier-manager.h"
#include "context-manager.h"
#include "threads.h"

namespace ns3 {

#define B_EMPTY_LOWER_BOUND 10 // TODO: Check this value, possibly tune it!
#define B_EMPTY_UPPER_BOUND 100000 // TODO: Check this value, possibly tune
    it!

TypeId
DecisionMaker::GetTypeId (void)
{
  static TypeId tid = TypeId ("ns3::DecisionMaker")
      .SetParent<Object> ()
      .AddConstructor<DecisionMaker> ()
      ;
    return tid;
}

DecisionMaker::DecisionMaker ()
{
  Ptr<CarrierManager> m_carrierManager = CreateObject <CarrierManager> ();
  AggregateObject(m_carrierManager);
  Ptr<ContextManager> m_contextManager = CreateObject<ContextManager> ();
  AggregateObject (m_contextManager);

  m_bEmptyRate = -1; // Unitialized value is -1

}

DecisionMaker::~DecisionMaker ()
{
}


DecisionType
NextHopDecision::GetDecisionType ()
{
  return m_type;
}

Ipv4Address
NextHopDecision::GetNextHopAddress ()
{
  return m_nextHop;
}


void
NextHopDecision::SetDecisionType (DecisionType dt)
```

```
{
  m_type = dt;
}

void
NextHopDecision::SetDestinationAddress (Ipv4Address addr)
{
  m_destination = addr;
}

Ipv4Address
NextHopDecision::GetDestinationAddress ()
{
  return m_destination;
}

void
NextHopDecision::SetNextHopAddress (Ipv4Address addr)
{
  m_nextHop = addr;
}

// Main decider function
NextHopDecision
DecisionMaker::DecideNextHop (Ptr<Packet> packet)
{
  /* Get the overlay message header */
  OverlayMessage om;
  packet->PeekHeader(om);

  /* Get resource manager */
  Ptr<ResourceManager> m_resourceManager = this->GetObject<ResourceManager>
      ();

  /* Get local address */
  Ipv4Address localaddr = m_resourceManager->GetLocalAddress ();

  /* Get Carrier manager */
  Ptr<CarrierManager> m_carrierManager = this->GetObject<CarrierManager> ();

  /* Destination of this packet */
  Ipv4Address destination = om.GetDestinationAddress ();

  /* Are we a carrier  */
  bool isCarrier = false;
  isCarrier = m_carrierManager->IsCarrier(localaddr, destination);

  /* Create variable for next hop decision */
  NextHopDecision nh;
  nh.SetDestinationAddress (destination);

  /* Check if we are at the destination */
  if (destination == localaddr)
  {
    nh.SetDecisionType(LOCAL_APP);
  }
```

```
else
{
  /* First look up the route in the routing table (do a call to
      ResourceManager) */
  Ipv4Address nextHop = m_resourceManager->LookupRoute (destination);

  /* If there exist a route: */
  if (nextHop != Ipv4Address("0.0.0.0"))
  {
    /* 1. If there is a queue, through another carrier, it means we are
        getting close to the destination, but the carrier might be going
        away.
       In this case, it is better to postpone the buffer emptying, and
           wait until we have one hop away from the CCC..
    */
    if (m_carrierManager->GetStrategyType () == CARRIER_STATIC
        && isCarrier && m_carrierManager->IsCarrier(nextHop, destination))
            {

      nh.SetDecisionType (LOCAL_BUFFER);
      StopEmptyAllBuffers (); //
    }
    /* 2. There exist a route, first check the link status..*/
    else if (!IsMacQueueFilling () and IsMacAddressExisting (nextHop)) {
      nh.SetDecisionType (OTHER_NODE);
      nh.SetNextHopAddress (nextHop);
    } /* 3. So the mac queue is filling, this means the thread for buffer
        emptying is postponed/stopped depending on the strategy. Destine
        for local buffer */
    else
      nh.SetDecisionType (LOCAL_BUFFER);
  }

  /* If no route exist, but we are a carrier, then we should carry! */
  else if (isCarrier)
  {
    nh.SetDecisionType (LOCAL_BUFFER);
  }

  /* Well, we should do something smart, because no route exist */
  else
  {
    if (m_carrierManager->GetStrategyType () == CARRIER_STATIC) //
        CARRIER_STATIC = Static-Dts
      { /* 1. Static carrier strategy */

        Ipv4Address carrier = m_carrierManager-> GetBestCarrier (
            destination);
        // Find nexthop to that carrier
        nextHop = m_resourceManager->LookupRoute (carrier);

        if (carrier != Ipv4Address("0.0.0.0") and !IsMacQueueFilling ()
            and IsMacAddressExisting (nextHop))
          {
            nh.SetDecisionType (OTHER_NODE);
            nh.SetNextHopAddress (nextHop);
          }
```

```
      /* No carrier, maybe see if we can do something smart?*/
      else
        {
          /* This counts as a strategy I.e., the "Previous Nexthop
              Strategy"*/
          NextHopDecision nexthopCache = m_resourceManager->
              GetDecisionCache (destination);
          Ipv4Address nextHop = m_resourceManager->LookupRoute (
              nexthopCache.GetNextHopAddress ());

          // If there exist a route:
          if (nextHop != Ipv4Address("0.0.0.0") and !IsMacQueueFilling
              () and IsMacAddressExisting (nextHop))
            {
              nh.SetDecisionType (OTHER_NODE);
              nh.SetNextHopAddress (nextHop);
            }
          else
            {
              nh.SetDecisionType (LOCAL_BUFFER);
            }
        }
    }
  else if (m_carrierManager->GetStrategyType () ==
      CARRIER_DELIVERY_PROBABILITY) // CARRIER_DELIVERY_PROBABILITY =
      DSMC
    {/* 2. Delivery probability strategy */

      /* If we are a carrier, and we meet another carrier that just
          departed the destination,
         that node is likly to have higher delivery probabilty than this
              node. However, we
         do not want to forward to that node, because that will lead to
             loops.
         We therefore buffer the message if we suspect we are a carrier
             for this destination.
      */
      if (m_carrierManager->CheckCouldBeCarrier(destination)) {

        nh.SetDecisionType (LOCAL_BUFFER);
        StopEmptyAllBuffers ();

      } else {

        Ipv4Address carrier = m_carrierManager->GetBestCarrier(
            destination);
        nextHop = m_resourceManager->LookupRoute (carrier);

        if (carrier != Ipv4Address("0.0.0.0")
            && nextHop != Ipv4Address("0.0.0.0")
            && !IsMacQueueFilling () && IsMacAddressExisting (nextHop))
          {
            nh.SetDecisionType (OTHER_NODE);
            nh.SetNextHopAddress (nextHop);
          }
        else {
          nh.SetDecisionType (LOCAL_BUFFER);
```

```
            }
          }
        }
    }
  }

  /* Pass decision to ResourceManager only if passed to somewhere else */
  if (nh.GetDecisionType () == OTHER_NODE) {
    m_resourceManager->UpdateNextHopDecision (om.GetDestinationAddress (),
        nh);
  }

  return nh;
} // End of DecideNextHop

void
DecisionMaker::NotifyRouteChange ()
{
  std::vector<Ptr <DtsStream> > streams = this->GetObject<ResourceManager>
      ()->GetDtsStreamVector ();

  if (streams.size () <= 0)
    return;

  // For each stream, check if we should change decission, given the new
      route table..
  for (std::vector<Ptr <DtsStream> >::iterator it = streams.begin (); it !=
      streams.end (); it++)
  {
    Ipv4Address nextHop = this->GetObject<ResourceManager> ()->LookupRoute
        ((*it)->GetDestinationAddress ());
    /* 1. If there is a route to the destination, obviously send the
        buffered data by starting the buffer empty thread */
    if (nextHop != Ipv4Address("0.0.0.0")) {
      GetObject<DtsOverlay> ()->StartEmptyBuffer ((*it));
    }
    else if (this->GetObject<CarrierManager> ()->GetStrategyType () ==
        CARRIER_DELIVERY_PROBABILITY &&
             this->GetObject<CarrierManager> ()->CheckCouldBeCarrier((*it)->
                 GetDestinationAddress()))
      { /* 1.5 Stop emptying buffer because we are still carrying (DSMC) */
      GetObject<DtsOverlay> ()->StopEmptyBuffer ((*it));
    }
    else
    { /* 2. If a new carrier has arrived, send the buffered data to that
        carrier */
      Ipv4Address carrier = this->GetObject<CarrierManager> ()->
          GetBestCarrier ((*it)->GetDestinationAddress());
      Ipv4Address nextHop = this->GetObject<ResourceManager> ()->LookupRoute
           (carrier);

      if (nextHop != Ipv4Address("0.0.0.0")
          && ! this->GetObject<CarrierManager> ()->
          IsCarrier(this->GetObject<ResourceManager> ()->GetLocalAddress ()
                    ,(*it)->GetDestinationAddress() ))
        {
        GetObject<DtsOverlay> ()->StartEmptyBuffer ((*it));
```

```
      }
      else /* No eiglible carrier exist, stop the buffer from emptying */
        GetObject<DtsOverlay> ()->StopEmptyBuffer ((*it));
    }
  }
} // End of NotifyRouteChange ()


double
DecisionMaker::DecideBufferEmptyRate () {

  if (m_bEmptyRate == -1) // We need to decide inital start rate ourselves.
      -1 if not even set, or if it has been reset.
    /* Startrate algorithm:*/
  {
    /* 1. Get average packet size in the buffer */
    int packet_count = this->GetObject<ResourceManager> ()->
        GetQueuedPacketCount ();
    int buffer_size = this->GetObject<ResourceManager> ()->GetQueuedBytes ()
        ;
    int average_size;

    if (buffer_size == 0 || packet_count == 0)
      average_size = 1400; // If we do not have any average size yet, we set
          it to MTU. TODO: Make dynamic
    else
      average_size = (buffer_size / packet_count);

    /* 2. Get available link, assume 11 Mbps link
     * http://en.wikipedia.org/wiki/IEEE_802.11b-1999 says we can expect 7.1
         Mbps UDP
     * Thus, we can assume 7.1 / 8 = 0.887500 Mbytes/s as an upper bound.
     *
     * Since the algorithm is dynamic, this start value is not that
        important. The rate wshould
     * anyhow converge to an optimal value
     *
     * TODO: Make dynamic this upper bound dynamic.
     * */
    int available_bw = (887500 - this->GetObject<ResourceManager> ()->
        GetMACBandwidthConsumption ());

    /* 3. Calculate the appropriate value */
    if (available_bw > 0)
    {
      m_bEmptyRate = (available_bw / average_size);
    }
    else // To avoid illegal arithmetic operations:
    {
      std::cout << "WARNING:␣No␣available␣bandwidth␣for␣buffer␣emptying:␣"
          <<  Simulator::Now () << "␣" << this->GetObject<ResourceManager>
          ()->GetLocalAddress () << std::endl;
      m_bEmptyRate = B_EMPTY_LOWER_BOUND; //pps;
    }

  }
```

```
  /* Buffer empty strategy 1: */
  if (this->GetObject<ResourceManager> ()->GetMacQueueSize () >= 1 &&
      m_bEmptyRate > B_EMPTY_LOWER_BOUND) { // TODO: CHECK THIS LIMIT
    m_bEmptyRate = (m_bEmptyRate / 2); // TODO: CHECK THIS LIMIT
  }
  else {
    m_bEmptyRate = m_bEmptyRate + 1;
  }

  if (m_bEmptyRate > B_EMPTY_UPPER_BOUND)
  {
    std::cout << "WARNING: Buffer empty rate to high, possibly circular
        behaviour " <<  Simulator::Now () << std::endl;
  }
  return (1.0/m_bEmptyRate);
}

void
DecisionMaker::StopEmptyAllBuffers ()
{
  //std::cout << "STOP_EMPTY_BUFFER: " << Simulator::Now () << " " << this->
      GetObject<ResourceManager> ()->GetLocalAddress () << std::endl;
  std::vector<Ptr <DtsStream> > streams = this->GetObject<ResourceManager>
      ()->GetDtsStreamVector ();

  for (std::vector<Ptr <DtsStream> >::iterator it = streams.begin (); it !=
      streams.end (); it++)
  {
    GetObject<DtsOverlay> ()->StopEmptyBuffer ((*it));
  }
}


bool
DecisionMaker::IsMacAddressExisting (Ipv4Address address)
{
  /** We now want to check if ARP has registered an MAC address for the IP
      address
   * 1. Check if exist. */
  if (GetObject<ResourceManager> ()->MACAddressExists (address)) {
    /** 1.1 Check if alive */
    if (GetObject<ResourceManager> ()->MACAddressARPAlive (address))
      return true;
    /** 1.2 If not alive, postpone! */
    else {
      GetObject<DtsOverlay> ()->PlayTableTennis (address);
      StopEmptyAllBuffers (); // TODO: Assure no starvation by "waking" up
          later in time to check if the links has not improved
      return false;
    }
  }
  /** 2. If not existing, we initate a new ARP request by playing table
      tennis, and stopping (postponing) buffer emptying. */
  else
  {
    GetObject<DtsOverlay> ()->PlayTableTennis (address);
    StopEmptyAllBuffers (); // TODO: Assure no starvation by "waking" up
```

```
        later in time to check if the links has not improved
    return false;
  }
 }

bool
DecisionMaker::IsMacQueueFilling ()
{
  if (this->GetObject<ResourceManager> ()->GetLinkAdapt () && this->
      GetObject<ResourceManager> ()->GetMacQueueSize () >= 75) // TODO:
      CHECK THIS LIMIT
  {
    StopEmptyAllBuffers (); // TODO: Assure no starvation by "waking" up
        later in time to check if the links has not improved
    return true;
  }
  else
    return false;
}

} // namespace ns3
```

Listing B.2: Code for Resource Manager

```
/* -*- Mode:C++; c-file-style:"gnu"; indent-tabs-mode:nil; -*- */
/*
 * Copyright (c) 2010 Morten Lindeberg
 *
 * This program is free software; you can redistribute it and/or modify
 * it under the terms of the GNU General Public License version 2 as
 * published by the Free Software Foundation;
 *
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with this program; if not, write to the Free Software
 * Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA
 *
 * Author: Morten Lindeberg (mglindeb@ifi.uio.no)
 */
#include "ns3/resource-manager.h"
#include "ns3/dca-txop.h"
#include "ns3/udp-header.h"
#include "ns3/llc-snap-header.h"
#include "ns3/arp-l3-protocol.h"
#include "ns3/ipv4-l3-protocol.h"
#include "ns3/ipv4-interface.h"
#include "ns3/arp-cache.h"

#include "ns3/decision-maker.h"
#include "ns3/dts-overlay.h"
#include "ns3/dts-stream.h"
#include "carrier-manager.h"
```

```
namespace ns3 {

TypeId
ResourceManager::GetTypeId (void)
{
  static TypeId tid = TypeId ("ns3::ResourceManager")
    .SetParent<Object> ()
    .AddConstructor<ResourceManager> ();
  return tid;
}

ResourceManager::ResourceManager ()
{
  m_resourceMonitor = CreateObject<ResourceMonitor> ();
  AggregateObject (m_resourceMonitor);
}

ResourceManager::~ResourceManager ()
{
}


void
ResourceManager::UpdateRoutingTable (long timestamp, std::vector<
    GenericRoutingTableEntry> r_entries) // TODO: Timestamp not nessecary!
{
  m_routeEntries = r_entries;

  // Notify decision maker:
  GetObject<DecisionMaker> ()->NotifyRouteChange ();
}

std::vector<GenericRoutingTableEntry>
ResourceManager::GetRouteEntries()
{
  return m_routeEntries;
}

uint32_t
ResourceManager::GetOneHopNeighborCount()
{
  uint32_t one_hop_cnt = 0;
  for (std::vector<GenericRoutingTableEntry>::iterator it =m_routeEntries.
      begin (); it != m_routeEntries.end (); it++)
    {
      if (it->GetDistance()==1)
        one_hop_cnt++;
    }
  return one_hop_cnt;
}



void
ResourceManager::UpdateNextHopDecision (Ipv4Address destination,
    NextHopDecision nh)
{
```

```
  /* Find relevant stream (if exist in streams vector) */
  for (std::vector<Ptr <DtsStream> >::iterator it = m_streams.begin (); it
      != m_streams.end (); it++)
  {
    if ( (*it)->GetDestinationAddress () == destination) {
      (*it)->AddDecision (nh);
      return;
    }
  }

  // Should never be reached. TODO: Use ASSERT?
  std::cout << "ERROR_ResourceManager::UpdateNextHopDecision:_stream_does_
      not_exist!" << std::endl;
  return;
}


NextHopDecision
ResourceManager::GetDecisionCache (Ipv4Address destination)
{
  /* Find relevant stream (if exist in streams vector) */
  for (std::vector<Ptr <DtsStream> >::iterator it = m_streams.begin (); it
      != m_streams.end (); it++)
  {
    if ((*it)->GetDestinationAddress () == destination)
      return (*it)->GetDecisionCache ();
  }

  // Should never be reached. TODO: Use ASSERT?
  std::cout << "ERROR_ResourceManager::GetDecisionCache:_stream_does_not_
      exist!" << std::endl;
  return NextHopDecision ();
}


Ptr<DtsStream>
ResourceManager::GetDtsStream (Ptr <Packet> packet)
{
  OverlayMessage om;
  packet->PeekHeader(om);

  // Search through the existing ones.
  for (std::vector<Ptr <DtsStream> >::iterator it = m_streams.begin (); it
      != m_streams.end (); it++)
  {
    /* TODO: Add e.g., filename to allow mulitple streams to one destination
        with different properties (not treated as the same) */
    if ((*it)->GetDestinationAddress () == om.GetDestinationAddress ()) {
      return *it;
    }
  }

  // It did not exist, we return a new one!
  Ptr<DtsStream> dtsStream = CreateObject<DtsStream> ();
  dtsStream->SetDestinationAddress (om.GetDestinationAddress ());
  dtsStream->SetLocalAddress (GetLocalAddress ());
  m_streams.push_back(dtsStream);
```

```
  return dtsStream;
}

std::vector<Ptr <DtsStream> >
ResourceManager::GetDtsStreamVector ()
{
  return m_streams;
}

Ipv4Address
ResourceManager::LookupRoute (Ipv4Address destination)
{
  if (m_routeEntries.empty () || destination == Ipv4Address("0.0.0.0"))
  {
    return Ipv4Address("0.0.0.0");
  }

  for (std::vector<GenericRoutingTableEntry>::iterator it = m_routeEntries.
      begin (); it != m_routeEntries.end (); it++)
  {
    if (destination == it->GetDestAddr ()) {
      return it->GetNextAddr ();
    }
  }

  return Ipv4Address("0.0.0.0");
}

int
ResourceManager::GetQueuedPacketCount ()
{
  int packet_count = 0;

  // Search through the existing traffic streams
  for (std::vector<Ptr <DtsStream> >::iterator it = m_streams.begin (); it
      != m_streams.end (); it++)
  {
    packet_count = packet_count + (*it)->GetBufferManager ()->
        GetQueuedPacketCount ();
  }

  return packet_count;
}


int
ResourceManager::GetDroppedQueuedPacketCount ()
{
  int dropped_count = 0;

  // Search through the existing traffic streams
  for (std::vector<Ptr <DtsStream> >::iterator it = m_streams.begin (); it
      != m_streams.end (); it++)
  {
    dropped_count = dropped_count + (*it)->GetBufferManager ()->
        GetDroppedPacketCount ();
```

```
  }

  return dropped_count;
}


int
ResourceManager::GetQueuedBytes ()
{
  int byte_size = 0;

  // Search through the existing traffic streams
  for (std::vector<Ptr <DtsStream> >::iterator it = m_streams.begin (); it
      != m_streams.end (); it++)
  {
    byte_size = byte_size + (*it)->GetBufferManager ()->GetQueuedBytes ();
  }

  return byte_size;
}


std::vector<GenericRoutingTableEntry>
ResourceManager::GetNeighborNodes ()
{
  std::vector<GenericRoutingTableEntry> out;

  for (std::vector<GenericRoutingTableEntry>::iterator it = m_routeEntries.
      begin (); it != m_routeEntries.end (); it++)
    {
      if (it->GetDistance () <= 1)
        out.push_back (*it);
    }
  return out;
}


bool
ResourceManager::IsCarrier(Ipv4Address address)
{
  Ipv4Address mask = address.CombineMask (Ipv4Mask("0.0.255.0"));
  if (mask == Ipv4Address ("0.0.2.0"))
    return true;
  return false;
}

std::vector<GenericRoutingTableEntry>
ResourceManager::GetCarrierNodes ()
{
  std::vector<GenericRoutingTableEntry> out;

  for (std::vector<GenericRoutingTableEntry>::iterator it = m_routeEntries.
      begin (); it != m_routeEntries.end (); it++)
  {
    Ipv4Address mask = it->GetDestAddr ().CombineMask (Ipv4Mask("0.0.255.0")
        );
    if (mask == Ipv4Address ("0.0.2.0"))
```

```
      out.push_back (*it);
  }
  return out;
}

Ipv4Address
ResourceManager::GetClosestCarrier (Ipv4Address dest)
{
  uint32_t hops = 9999; // keeps track of the lowest hop count TODO: Set a
      better value
  Ipv4Address closest = Ipv4Address("0.0.0.0");; // keeps track of the next
      hop address of the destination node with lowest hop count

  /* Get Carrier manager */
  Ptr<CarrierManager> m_carrierManager = this->GetObject<CarrierManager> ();


  for (std::vector<GenericRoutingTableEntry>::iterator it = m_routeEntries.
      begin (); it != m_routeEntries.end (); it++)
    {
      if (m_carrierManager->IsCarrier(it->GetDestAddr(), dest))
        {
          if (it->GetDistance () < hops) // TODO: THIS MUST / SHOULD BE
              OPTIMIZED!
          {
            closest = it->GetDestAddr ();//it->GetNextAddr ();
            hops = it->GetDistance ();
          }
        }
    }
  if (hops < 9999)
    {
      return closest;
    }
  else
    {
      return Ipv4Address ("0.0.0.0");
    }
}

uint32_t
ResourceManager::GetDistanceHops(Ipv4Address addr) //FIX used?
{
  for (std::vector<GenericRoutingTableEntry>::iterator it = m_routeEntries.
      begin (); it != m_routeEntries.end (); it++)
    if (it->GetDestAddr() == addr)
      return it->GetDistance();
  return 9999;
}

Ipv4Address
ResourceManager::GetLocalAddress ()
{
  if (this == 0)
  {
    std::cout << "ERROR: WE ResourceManager CALLED BUT DOES NOT EXIST!" <<
        std::endl;
```

```
    return Ipv4Address("0.0.0.0");
  }
  return m_localaddr;
}

void
ResourceManager::SetLocalAddress (Ipv4Address localaddrValue)
{
  m_localaddr = localaddrValue;
  m_resourceMonitor->SetLocalAddress(localaddrValue);
}

void
ResourceManager::InsertMACTuple (long timestamp, int nodeid, int size)
{
  m_resourceMonitor->InsertQuery1Tuple(timestamp, nodeid, size);
}

int
ResourceManager::GetMACBandwidthConsumption ()
{
  return m_resourceMonitor->GetQuery1Result();
}

void
ResourceManager::NotifyMACFailure (long timestamp)
{
  // Note: This should be done in DecisionMaker:
  Ptr<Packet> packet = m_dca->GetLostPacket ();

  // Remove LlcSnapHeader identifying protocols by Ethernet type field
      values (IEEE 802.2)
  LlcSnapHeader llcSnapHeader;
  packet->RemoveHeader (llcSnapHeader);

  if (llcSnapHeader.GetType () != (0x800)) // Return if packet is not a IP
      packet (meaning ARP)! NOTE 0X800 is known for Ether Type IP.
    return;

  // Check that the frame is not part of a fragmented packet
  WifiMacHeader wifiMacHeader;
  packet-> PeekHeader (wifiMacHeader);

  if (wifiMacHeader.IsMoreFragments ()) {
    std::cout << "IS_MORE_FRAGMENTS: " << timestamp << " " << std::endl;
    return;
  }

  if (wifiMacHeader.GetFragmentNumber() > 1)
  {
    std::cout << "FRAGMENT_NUMBER: " << timestamp << " " <<  wifiMacHeader.
        GetFragmentNumber () << " "<< wifiMacHeader.GetSize() << "("<<
        wifiMacHeader.GetSequenceNumber () << ")" << std::endl;
    return;
  }

  // Now remove IP header
```

```
  Ipv4Header ipHeader;
  packet->RemoveHeader (ipHeader);

  // Trim any remaining frame padding from underlying devices
  if (ipHeader.GetPayloadSize () < packet->GetSize ())
  {
    packet->RemoveAtEnd (packet->GetSize () - ipHeader.GetPayloadSize ());
  }

  if (ipHeader.GetDestination() == Ipv4Address ("0.0.0.0"))
    std::cout << "WE HAVE OUR ERROR RIGHT HERE" << std::endl;

  // Now remove UDP header
  UdpHeader udpHeader;
  packet->RemoveHeader(udpHeader);

  // Check that it is a overlay message:
  if (udpHeader.GetDestinationPort () != 12345) {
      std::cout << "MAC_RETURN: " << timestamp << " dropped lost packet that
          was not a OverlayMessage " << std::endl;
      return;
  }

  OverlayMessage om;
  packet->PeekHeader(om);

  //std::cout << "HANDLE_MAC_FAILURE: " << Simulator::Now () << " from=" <<
      GetLocalAddress () << ", to="
  //          << om.GetDestinationAddress () << " (Routing protocol suggest:
       " << LookupRoute (om.GetDestinationAddress ())
  //          << ","<< GetMacQueueSize ()  << ")" << std::endl;

  // Strategy 1:
  //GetObject<DtsOverlay> ()->HandlePacket(packet); // Send it back to the
      overlay
  GetObject<DtsOverlay> ()->StopEmptyBuffer(GetDtsStream (packet)); // Stop
      empty all buffers until route change!
  GetDtsStream (packet)->InsertMessage (packet);
}

Ipv4Address
ResourceManager::GetNewestArrivedCarrier ()
{
  return m_resourceMonitor->GetNewestArrivedCarrier ();
}

uint32_t
ResourceManager::GetMacQueueSize ()
{
  return m_dca->GetQueueSize ();
}

uint32_t
ResourceManager::GetMaxMacQueueSize ()
{
  return m_dca->GetMaxQueueSize ();
}
```

```
void
ResourceManager::EnableLinkAdapt ()
{
  //std::cout << "ENABLE_LINK_ADAPT: " << m_localaddr << std::endl;
  m_linkAdapt = true;
}

bool
ResourceManager::GetLinkAdapt ()
{
  return m_linkAdapt;
}

void
ResourceManager::SetDcaTxop (Ptr<DcaTxop> dca) // TODO: Remove!
{
  m_dca = dca;
}

void
ResourceManager::EnableARPAdapt ()
{
  //std::cout << "ENABLE_ARP_ADAPT: " << m_localaddr << std::endl;
  m_arpAdapt = true;
}

bool
ResourceManager::GetARPAdapt ()
{
  return m_arpAdapt;
}


bool
ResourceManager::MACAddressExists (Ipv4Address address)
{
  if (!m_arpAdapt)
    return true;

  Ptr<ArpCache> arp = GetObject<DtsOverlay>()->GetObject<Ipv4L3Protocol>()->
      GetInterface(1)->GetArpCache();

  if (arp->Lookup (address) == NULL)
    return false;
  return true; //(arp->Lookup (address)->IsAlive ());
}

bool
ResourceManager::MACAddressARPAlive (Ipv4Address address)
{
  if (!m_arpAdapt)
    return true;

  Ptr<ArpCache> arp = GetObject<DtsOverlay>()->GetObject<Ipv4L3Protocol>()->
      GetInterface(1)->GetArpCache();
  return (arp->Lookup (address)->IsAlive ());
```

```
}

} // namespace ns3
```

Listing B.3: Code for Carrier Manager

```
/* -*- Mode:C++; c-file-style:"gnu"; indent-tabs-mode:nil; -*- */
/*
 * Copyright (c) 2011 Jan Erik Haavet
 *
 * This program is free software; you can redistribute it and/or modify
 * it under the terms of the GNU General Public License version 2 as
 * published by the Free Software Foundation;
 *
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with this program; if not, write to the Free Software
 * Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA
 *
 * Author: Jan Erik Haavet (janehaa@ifi.uio.no)
 */
#include <iostream>
#include "ns3/ipv4-address.h"
#include "ns3/ipv4.h"
#include "ns3/simulator.h"
#include "ns3/double.h"
#include "ns3/uinteger.h"
#include "threads.h"

#include "carrier-manager.h"

namespace ns3 {


NS_OBJECT_ENSURE_REGISTERED (CarrierManager);

TypeId
CarrierManager::GetTypeId (void)
{
  static TypeId tid = TypeId ("ns3::CarrierManager")
    .SetParent<Object> ()
    .AddConstructor<CarrierManager> ()
    .AddAttribute ("CarrierStrategyType",
                   "Type of strategy used for finding carriers",
                   UintegerValue(0),
                   MakeUintegerAccessor (&CarrierManager::m_strategyType),
                   MakeUintegerChecker<uint32_t> ())
      ;
    return tid;
}

CarrierManager::CarrierManager ()
{
```

```
}

CarrierManager::~CarrierManager ()
{
}

Ipv4Address
CarrierManager::GetBestCarrier(Ipv4Address dest)
{
  switch (m_strategyType)
    {
    case CARRIER_STATIC:
      return this->GetObject<ResourceManager> ()->GetClosestCarrier(dest);
      break;
    case CARRIER_DELIVERY_PROBABILITY:
      return this->GetObject<ResourceManager> ()->GetObject<ContextManager>
          ()->GetBestCarrier(dest);
    }
  return Ipv4Address("0.0.0.0");
}

bool
CarrierManager::IsCarrier(Ipv4Address addr, Ipv4Address dest)
{
  switch (m_strategyType)
  {
  case CARRIER_STATIC:
    {
      Ipv4Address mask = addr.CombineMask (Ipv4Mask("0.0.255.0"));
      if (mask == Ipv4Address("0.0.2.0"))
        return true;
      break;
    }
  case CARRIER_DELIVERY_PROBABILITY:
      return (addr != Ipv4Address("0.0.0.0") && this->GetObject<
          ResourceManager> ()->GetObject<ContextManager> ()->GetBestCarrier(
          dest) == addr);
  }

  return false;
}

bool
CarrierManager::CheckCouldBeCarrier(Ipv4Address destination)
{
  switch (m_strategyType)
  {
  case CARRIER_STATIC:
    {
      return IsCarrier(this->GetObject<ResourceManager> ()->GetLocalAddress
          (), destination);
      break;
    }
  case CARRIER_DELIVERY_PROBABILITY:
    return this->GetObject<ResourceManager> ()->GetObject<ContextManager> ()
        ->CheckCouldBeCarrier(destination);
  }
```

```
  return false;
}

void
CarrierManager::SetStrategyType(uint32_t type)
{
  m_strategyType = type;
}

uint32_t
CarrierManager::GetStrategyType()
{
  return m_strategyType;
}

}
```

Listing B.4: Code for Context Manager

```
/* -*- Mode:C++; c-file-style:"gnu"; indent-tabs-mode:nil; -*- */
/*
 * Copyright (c) 2011 Jan Erik Haavet
 *
 * This program is free software; you can redistribute it and/or modify
 * it under the terms of the GNU General Public License version 2 as
 * published by the Free Software Foundation;
 *
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with this program; if not, write to the Free Software
 * Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA
 *
 * Author: Jan Erik Haavet (janehaa@ifi.uio.no)
 */

#include <iostream>
#include "context-manager.h"
#include "ns3/double.h"

namespace ns3 {

NS_OBJECT_ENSURE_REGISTERED (ContextManager);

  TypeId
  ContextManager::GetTypeId (void)
  {
    static TypeId tid = TypeId ("ns3::ContextManager")
      .SetParent<Object> ()
      .AddConstructor<ContextManager> ()
      .AddAttribute ("AlphaVariable",
                     "Factor_at_which_contact_probabilities_increase/
                         decrease",
                     DoubleValue(0.02),
```

```
                    MakeDoubleAccessor (&ContextManager::m_alpha_constant),
                    MakeDoubleChecker<double> ())

    ;
  return tid;
}

ContextManager::ContextManager ()
{
  //SetAttribute ("ns3::ContextManager::AlphaVariable", DoubleValue(0.02))
      ; // Aprox 500 Mb
  //m_alpha_constant = 0.005; //NOTE: for scaled scenario 2  = 0.005;
      normal scenario 1 = 0.02;
  m_localaddr = Ipv4Address("0.0.0.0");

  significance_weight[CONTACT_PROB] = 1.0;
}
ContextManager::~ContextManager ()
{
}

Ipv4Address
ContextManager::GetLocalAddr()
{
  return m_localaddr;
}

void
ContextManager::SetLocalAddr(Ipv4Address localaddr)
{
  m_localaddr = localaddr;
}


/*** DELIVERY PROBABILITY ***/

Ipv4Address
DeliveryProbabilityEntry::GetDestAddr ()
{
  return m_destAddr;
}

void DeliveryProbabilityEntry::SetDestAddr (Ipv4Address destAddr)
{
  m_destAddr = destAddr;
}

Ipv4Address
DeliveryProbabilityEntry::GetBestCarrierAddr()
{
  return m_bestCarrierAddr;
}

void
DeliveryProbabilityEntry::SetBestCarrierAddr(Ipv4Address bestCarrierAddr)
{
  m_bestCarrierAddr = bestCarrierAddr;
```

```
}

double
DeliveryProbabilityEntry::GetBestCarrierDelProb()
{
  return m_bestCarrierDelProb;
}

void
DeliveryProbabilityEntry::SetBestCarrierDelProb(double bestCarrierDelProb)
{
  m_bestCarrierDelProb = bestCarrierDelProb;
}

std::map<Ipv4Address, DeliveryProbabilityEntry>
ContextManager::GetDeliveryProbabilityTable()
{
  return r_deliveryProbabilities;
}

Ipv4Address
ContextManager::GetBestCarrier(Ipv4Address dest)
{
  std::map<Ipv4Address, DeliveryProbabilityEntry>::iterator it;
  it = r_deliveryProbabilities.find(dest);
  if (it != r_deliveryProbabilities.end() && (*it).second.GetDestAddr() !=
      m_localaddr)
    return (*it).second.GetBestCarrierAddr();

  return Ipv4Address("0.0.0.0");
}

void
ContextManager::UpdateDeliveryProbablities(Ipv4Address dest, Ipv4Address
    possible_best_carrier, double del_prob)
{
  if (dest == m_localaddr) //avoid entry for yourself
    return;

  std::map<Ipv4Address, DeliveryProbabilityEntry>::iterator it;
  it = r_deliveryProbabilities.find(dest);
  if (it == r_deliveryProbabilities.end()) {//new entry

    r_deliveryProbabilities.insert(std::pair<Ipv4Address,
        DeliveryProbabilityEntry>
                                   (dest, DeliveryProbabilityEntry(dest,
                                       possible_best_carrier, del_prob)));

  } else { //we can try to update an entry
    if (((*it).second.GetBestCarrierAddr() == m_localaddr && del_prob > (*
        it).second.GetBestCarrierDelProb()) //only overwrite local
        forwarder if there is a better one
        || ((*it).second.GetBestCarrierAddr() != m_localaddr && del_prob
            >= (*it).second.GetBestCarrierDelProb()) //if previous best
            was not local, equal is enough (also more updated)
        ) {
      (*it).second.SetBestCarrierDelProb(del_prob);
```

```
      (*it).second.SetBestCarrierAddr(possible_best_carrier);
    }
  }
}

void
ContextManager::PrintDeliveryProbabilities()
{
  std::cout << "PrintDeliveryProbabilities:: local_addr: " << m_localaddr
      << std::endl;
  std::map<Ipv4Address, DeliveryProbabilityEntry>::iterator it;
  for (it = r_deliveryProbabilities.begin (); it !=
      r_deliveryProbabilities.end (); it++)
    std::cout << "  dest: " << (*it).second.GetDestAddr() << " best
        carrier: " << (*it).second.GetBestCarrierAddr()
              << " prob: " << (*it).second.GetBestCarrierDelProb() << std
                  ::endl;
}

void
ContextManager::CalcDeliveryProbabilities()
{
  //for each destination (met so far), combine context attributes to a
      delivery probability value
  Ipv4Address dest;
  std::map<Ipv4Address, ContactProbabiltyEntry>::iterator m_it;
  for (m_it = m_contacts.begin (); m_it != m_contacts.end (); m_it++) {
    dest = (*m_it).first;
    if (dest != m_localaddr) { //skip yourself!
      double del_prob = 0;

      //for each attribute
      for (int i=0; i<MAX_CONTEXT_TYPE; i++)
        {
          switch(i)
            {
            case CONTACT_PROB:
              del_prob += (*m_it).second.contact_prob *
                  significance_weight[CONTACT_PROB];
              break;
            }
        }

      //update of local delivery probabilities
      UpdateDeliveryProbablities(dest, m_localaddr, del_prob);
    }
  }
}

bool
ContextManager::CheckCouldBeCarrier(Ipv4Address destination)
{
  std::map<Ipv4Address, ContactProbabiltyEntry>::iterator m_it;
  m_it = m_contacts.find(destination);
  double del_prob_limit = 0.01;
  double del_prob = 0;
```

```
  if (m_it != m_contacts.end()) {
    //for each attribute
    for (int i=0; i<MAX_CONTEXT_TYPE; i++) {
      switch(i) {
      case CONTACT_PROB:
        del_prob += (*m_it).second.contact_prob * significance_weight[
            CONTACT_PROB];
        break;
      }
    }
  }

  return del_prob >= del_prob_limit;
}

void
ContextManager::AgeDeliveryProbabilities()
{
  double aging_factor = 0.95;
  std::map<Ipv4Address, DeliveryProbabilityEntry>::iterator it;
  for (it = r_deliveryProbabilities.begin (); it !=
      r_deliveryProbabilities.end (); it++) {
    (*it).second.SetBestCarrierDelProb ((*it).second.GetBestCarrierDelProb
        () * aging_factor);
  }
}


/** CONTACT PROBABILITY ATTRIBUTE **/

void
ContextManager::UpdateContactProbabilities(std::vector<
    GenericRoutingTableEntry> r_entries, int t_slot)
{
  /* Algorithm:
     If i meets j in this timeslot: Eij = (1-a)[Eij]old + a
     If they do not meet: Eij = (1-a)[Eij]
   */
  m_resourceManager = this->GetObject<ResourceManager> ();

  //update all contact probabilities for contacts in r_entries
  std::map<Ipv4Address, ContactProbabiltyEntry>::iterator m_it;
  for (std::vector<GenericRoutingTableEntry>::iterator it = r_entries.
      begin (); it != r_entries.end (); it++)
    {
      if (it->GetDistance() < 2) { //Only direct contact
        if ((m_it = m_contacts.find(it->GetDestAddr())) != m_contacts.end
            ())
          {
            (*m_it).second.contact_prob = (1-m_alpha_constant) * (*m_it).
                second.contact_prob + m_alpha_constant;
            (*m_it).second.time_slot = t_slot;
          }
        else
          {//new entry
            m_contacts[it->GetDestAddr()].contact_prob = 0.1; //inital
                value
```

```
            m_contacts[it->GetDestAddr()].time_slot = t_slot;
          }
      }
    }

  //update all contact probabilities for contacts not in r_entries
  // I.e., those entries that did not get update above.
  for (m_it = m_contacts.begin (); m_it != m_contacts.end (); m_it++)
    {
      if ((*m_it).second.time_slot != t_slot)
        {
          // NOTE: During runs, I got some strange values for contact_prob
               when it got very low.
          // I suspected that it was some form of underflow issue.
          // Hence, this test. It does not matter if contact_prob is 0.01
               or 0.0000001 anyway
          if ((*m_it).second.contact_prob < 0.01)
            (*m_it).second.contact_prob = 0.01;
          else
            (*m_it).second.contact_prob = (1-m_alpha_constant) * (*m_it).
                 second.contact_prob;
          (*m_it).second.time_slot = t_slot;
        }
    }
}

void ContextManager::PrintContactProbabilities() {
  std::cout << "ContactProbabilities:" << std::endl;
  std::map<Ipv4Address, ContactProbabiltyEntry>::iterator m_it;
  for (m_it = m_contacts.begin (); m_it != m_contacts.end (); m_it++)
    std::cout << "\tDest: " << (*m_it).first << ", Prob: " << (*m_it).
        second.contact_prob << ", Time slot: " << (*m_it).second.time_slot
         << std::endl;
}

/** NEXT ATTRIBUTE ... **/

}; // namespace ns3
```

Appendix C

DVD-ROM

We provide a DVD-ROM that includes: (1) The NS-3 code base with the Dts-Overlay and DSMC. (2) Scripts for starting simulation runs, calculating average results and generate gnuplot files for evaluation graphs.

C.1 Code

The code is found in the *code/ns-3.10/* folder relative to the root of the DVD. DSMC and Dts-Overlay can be found in *code/ns-3.10/src/overlay/dts-overlay/*. The code that differentiates a sending node (dts-trace-client) with a recieving node (dts-server) can be found in *code/ns-3.10/src/applications/*.

The simulation setup (dts-simulation), cross layering and exchange of delivery probabilities (cross-layer-exchange) can be found in *code/ns-3.10/scratch/dts/*.

C.2 Scripts

To start a simulation run of Dts-Overlay using DSMC in NS3, we have provided two scripts: The first is *code/ns-3.10/my_config_runner.py*, which runs all configurations for Scenario 1 (See Section 7.5). Each configuration is run 5 times and the results are placed in *code/ns-3.10/eval_runs/TODAY/sc1/configX*. *TODAY* is replaced with the day the script is run, i.e., 06_11_11, and *X* is replaced with the configuration number that is used. The second script is *code/ns-3.10/my_config_runner_scaled.py*, and does the exact same thing as the first script, except its for Scenario 2 and is stored in *code/ns-3.10/eval_runs/TODAY/sc2/configX*. In the delivered code, these scripts are set up to run all configurations possible for 5 runs. This can potentially take a long time (over 12 hours). To limit these

scripts to a single configuration, they need to be edited. There are instructions on how to limit the script to certain configurations in the script itself. (Also note that running all the configurations require quite a bit of storage space. Something like 20GB.)

The two scripts copy additional scripts from the *code/ns-3.10/eval_runs/scripts/* folder to the *configX* folder. These scripts are for calculations and producing gnuplot files for performance results. Specifically, the scripts that are copied into each are:

- RunAvg.class (java version 1.6.0_20): This script calculates the averages and standard deviations of PD, PB and PL (See Section 7.5). It takes two arguments: configuration number and number of runs. An example of a run in the config1 folder: java RunAvg 1 5
- get_delay.py (python version 2.6.5): This script scans simulation logs and retrieves the delay data from all runs in a configuration. It produces a gnuplot script with the following format: *plot_cdf_delay_cmp_confX.gp*, where *X* is the current configuration. It takes one argument, which is the configuration number. An example of a run in the config1 folder: python get_delay.py 1. The gnuplot script will produce a a graph that shows the CDF for all runs in that configuration.
- hist: this is a simple program that is used by get_delay.py to sort the delay data.

For clarification, we now provide a full example of how all scripts can be run to simulate different configuration runs and produce the data need to show evaluation results. This example can be found in listing C.1 and shows how to run all configurations for Scenario 1 and how to create graphs for PD and CDF. Additionally, it shows how to get the average results for PD, PB and PL.

Listing C.1: Scenario and Configuration Run

```
$
$cd code/ns-3.10/
$python my_config_runner.py
$cd eval_runs/07_11_11/sc1/config1
$gnuplot plot_r1.gp
$python get_delay.py 1
$gnuplot plot_cdf_delay_cmp_conf1.gp
$java RunAvg 1 5
```

If something should go wrong, it is possible to run: *./waf distclean* to clean and then *./waf* to rebuild NS3.